Master's Degree in Informatics Engineering
Dissertation

Network Softwarization for IACS Security Applications

Miguel Rosado Borges de Freitas

miguelbf@student.dei.uc.pt

Supervisor: Prof. Doutor Tiago Cruz
Co-Supervisor: Prof. Doutor Paulo Simões

Coimbra, Sunday 2nd September, 2018

FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

Over recent years, Industrial and Automation Control Systems (IACS) adopted in critical infrastructures have become more complex due to the increasing number of interconnected devices. Business goals to improve efficiency and productivity within critical infrastructures have connected the once isolated and trusted (due to obscurity) industrial control systems with external networks. This inter-connectivity disclosed and amplified the weaknesses of traditional IACS protocols, leading to a growing number of cyber-attacks specifically targeting critical infrastructures.
Enforcing security policies through the deployment of network intrusion and detection systems, honeypots (security probes) or other security mechanisms (such as uni-directional gateways) is hard to accomplish (and often error prone) in traditional network architectures. The network ecosystem is extremely complex, composed of several multi-vendor physical devices which are statically deployed in the network and configured on a per-device level. In the occurrence of a cyber-attack, the network cannot easily block unwanted traffic or evolve to new topologies. New ICT paradigms such as Software Defined Network (SDN) and Network Function Virtualization (NFV) are showing promising results in the cloud computing domain, providing innovative features for flexible and efficient management, monitoring and control of the network. The first (SDN) by allowing network programmability and protocol abstractions from a logically centralized location (the network controller). The later (NFV) by shifting network packet inspection from physical dedicated hardware to virtualized computational instances running in commercial-off-the-shelf hypervisors.
This thesis explores the synergies between SDN and NFV to apply a software defined security approach to IACS. An SDN based architecture is proposed and implemented for the easy deployment of containerized versions of typical IACS security probes. The proposed architecture explores the distributed nature of an SDN controller (ONOS) targeting performance and availability in a multi-tenancy network environment.

Keywords: Industrial Automation and Control systems, Software Defined Networking, Network Function Virtualization, Intrusion and Detection Systems, Network Honeypot, Data Diode, Multi-tenancy, Container based virtualization

Acknowledgement

I would like to express my sincere gratitude to both my supervisors, Prof.Doutor Tiago Cruz and Prof.Doutor Paulo Simões, for the enormous contribution and valuable discussions provided throughout the accomplishment of this dissertation. For the availability, development and implementation of the datacenter Environmental Monitoring Unit (EMU) and for providing the necessary conditions for a successful research effort.
To Luis Rosa for the guidance and continuous availability anytime a shortcoming was faced, a request had to be fulfilled or a complex problem needed to be tackled as a team. Thanks also for the development of the generic *IADS-Avro* library which greatly reduced the work required to have a working SDN event factory proof-of-concept.
Victor Graveto for the *Golang* primer necessary for the development of the SDN glue agent and global help during the project.
To Jorge Proença for the prompt response to any network/virtual machine (VM) tickets and for the work on the *RapidScada* HMI.
To Pedro Quitério for the joint work and last stage help in developing the IADS management web-interface.
To Leonardo Toledo for the all the help eliciting the IADS requirements.
To Rui Queiroz for the initial input regarding SDN/OpenFlow and for reviewing the state-of-the art chapter.
To Filipe Sequeira for proof-reading several sections of this thesis. To all of them for contributing to the spirit of the room G5.4 making it one of the best places to do research within LCT.

A worth mention to the Israel Electric Corporation (IEC) team (specially Ofer Bar) for providing the required hardware and help troubleshooting the replication of the SDN testbed for the final review of the ATENA project. To the ATENA H2020 Project (H2020-DS-2015-1 Project 700581), consortium and all its partners for funding this research.

Last words go to my family, to whom this thesis is dedicated. For all the support and perseverance during these though years - that I sincerely hope lead to complete career shift.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ACL	Access Control Lists.
AMQP	Advanced Message Queuing Protocol.
API	Application Programming Interface.
ARP	Address Resolution Protocol.
BDDP	Broadcast Domain Discovery Protocol.
BGP	Border Gateway Protocol.
CLI	Command-Line.
COTS	Commercial off-the-shelf.
DCS	Distributed Control System.
DDoS	Distributed Denial of Service.
DER	Distributed Energy Resources.
DMA	Direct Memory Access.
DOM	Document Object Model.
DoS	Denial of Service.
DPDK	Data Plane Development Kit.
DPI	Deep Packet Inspector.
DSCP	Differentiated Services Code Point.
EMU	Environment Monitoring Unit.
EV	Electric Vehicles.
FCA	Forensics and Compliance Auditing.
FPGA	Field-Programmable Gate Array.
HAZOP	Hazard and operability study.
HMI	Human Machine Interface.
I/O	Input/Output.
IaaS	Infrastructure-as-a-Service.
IACS	Industrial and Automation Control Systems.
IC	Integrated circuit.
ICT	Information and Communication Technology.
IDS	Intrusion Detection System.
IED	Intelligent Electronic Device.
ILP	Integer Linear Programming.
IoT	Internet of Things.
IP	Internet Protocol.
IPAM	IP Address Management.

LLDP	Link Layer Discovery Protocol.
LXC	Linux Containers.
MD-SAL	Model Driven Service Abstraction Layer.
MVC	Model-View-Controller.
MVVM	Model–View–Viewmodel.
NFV	Network Function Virtualization.
NIDS	Network Intrusion Detection System.
NOS	Network Operating System.
ONOS	Open Networking Operating System.
OSGi	Open Services Gateway initiative.
OVS	Open Virtual Switch.
PCE	Path Computation Element.
PDC	Power Distribution Center.
PID	Process ID.
PLC	Programmable Logic Controller.
PMU	Phasor Measurement Unit.
QoS	Quality-of-Service.
RERO	Release Early Release Often.
REST	Representational State Transfer.
RT	Real-Time.
RTOS	Real-Time Operating System.
RTSP	Rapid Spanning Tree Protocol.
RTU	Remote Terminal Unit.
SCADA	Supervisory Control and Data Acquisition.
SDECN	Software Defined Energy Communication Network.
SDN	Software Define Networking.
SIEM	Security Information and Event Management.
SNMP	Simple Network Management Protocol.
SPAN	OpenDayLight.
SPAN	Switch Port Analyzer.
SVM	Support Vector Machines.
TAP	Test Access Point.
TCAM	Ternary Content Addressable Memory.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
VFIO	Virtual Function I/O.
VLAN	Virtual Local Area Network.
VM	Virtual Machine.

vNF	Virtual Network Function.
WAN	Wide-Area Network.

Chapter 1

Introduction

1.1 Motivation

Today, the industrial automation and control systems (IACS) used to control and monitor mission critical applications such as the power grid, oil and gas distribution or chemicals manufacturing are turning ubiquitous. Current trends such as the "Industry 4.0" and the Internet of Things (IoT) are making traditional industrial control systems to move away from their original isolation concept and to get interconnected and exposed to external networks. These systems generate, process and exchange vast amounts of critical information, making them attractive targets for cyber based attacks. Cyber-attacks against IACS can result in much more disastrous consequences compared to other targets. The cyber-physical interdependencies in IACS give cyber-attacks the ability to disrupt nations' essential services, to cause physical damage and to even threaten human lives. The convergence between ICT and IACS greatly expanded the cyber-attack exploitation surface: not only the control instrumentation is accessible from corporative networks but used communication protocols are just minor revisions of legacy protocols so they can operate over the TCP/IP stack (without any improvements in terms of security and privacy). The Stuxnet attack in 2010 (Wired 2014) made the world aware of the vulnerabilities of IACS and led to a growing research interest in securing such systems.

Monitoring and deploying security devices for intrusion detection can no longer comply with traditional network architecture, as we move towards the IoT-generation of IACS and try to monitor every single point of the critical infrastructure. Traditional network architectures have the forwarding and control planes strongly coupled in each network device and require multiple physical deployments to monitor the whole control system infrastructure. Software Defined Networking (SDN), an architecture which aims at decoupling the control plane from the forwarding plane of network devices and promotes network flow programmability from a centralized location; has moved out of the research field and started to be used in real world implementations – pushed by internet giants and big telecom companies. Complemented with network function virtualization (NFV), SDN is being used by service providers to replace network functions implemented in legacy middle-boxes with virtual network functions operating in virtual machines (VMs) of their datacenter. While the first IACS-SDN turn-key solutions are starting to appear the synergies between SDN and NFV in the IACS field are yet to mature and be investigated. In distributed and ubiquitous IACS, taking advantage of both technologies seems to be a promising way of overcoming the current bottlenecks associated with monitoring the critical infrastructure.

1.2 Context

Considering today's highly distributed IACS, the ATENA H2020 project (ATENA 2017) aims at building upon the knowledge acquired in previous European Research activities (particularly from FP7 CockpitCI and MICIE projects) to push innovation through the exploitation of advanced features of ICT and Cyber Security so they can be adopted at the operational industrial level. The University of Coimbra (FCTUC) leads a work package in the ATENA project which proposes to tackle the challenges of monitoring (and securing) such high capillarity infrastructures with a distributed awareness approach. Hence, the adoption of Big Data-like data processing strategies (such as the use of Scalable Complex Event Processing), coupled with Fog-Computing mechanisms (supported by containerization) are envisioned as strategic assets in the architecture to be developed. The overall platform to be developed by the University of Coimbra in the ATENA context is called the *Intrusion and Anomaly Detection System* (IADS).

The ATENA distributed security awareness concept encompasses a Big Data SIEM (Security Information Event Management) capable of providing a source data-frame for forensics and auditing purposes, which also represents the central logical piece of the architecture. In the SIEM, machine learning algorithms predict the probabilities of the infrastructure being under a cyber-attack. This architecture strongly depends on security probes: to be able to accomplish big-data processing, distributed assets have to collect and forward monitoring data (or security events) to the SIEM. From this perspective, Software Defined Networking and Network Function Virtualization arise as auxiliary (and complementing) technologies to support the distributed monitoring infrastructure. While NFV is used to remove the complete dependency on physical probes moving some of them to the datacenter (hypervisors) as virtual network functions (VNFs), SDN programs the network flows (through flow rule deployments) to ensure the virtual security probes are able to operate. The application of NFV and SDN within the context of the IADS cyber-security constitutes one of the main innovations of the ATENA project.

1.3 Objectives

The nature of this thesis is twofold, since it has a strong **research component** as well as a strong **development and implementation component**. The main goal of this dissertation work is to propose and implement a high-availability and performant architecture for the deployment of IACS security probes though the combination of SDN and NFV technologies. More precisely, this work focuses on porting existing security IACS mechanisms - intrusion detection systems (IDS), honeypots or unidirectional gateways (data diodes) - to a new, software defined, security paradigm. Security probes once requiring dedicated hardware need to be ported to lightweight containers and deployed in a shared virtualization infrastructure. The deployment of such virtual probes should be managed by means of an SDN controller that orchestrates deployments and programs the underlying network through the installation of flow rules, e.g. using the OpenFlow protocol. More precisely, the system has to install rules to provide copies of host traffic (intrusion detection systems), redirect traffic flows (honeypot) or block traffic in certain directions (data diode). The system should be managed by a logically centralized entity, assuring both the virtual probes (containers) and the network hosts share the same network topology and are able to communicate.

Regarding the **research component**, it is fundamental to understand if the supporting technologies can be used in the IACS domain and respect some of their fundamental requirements. For example, in current IACS, multiple operators are involved in managing the network and the process control operations (e.g. subcontractors). The need for multi-tenancy and the stratification of user roles is a fundamental requirement. It is also important to understand how the SDN architecture can be used to improve the resilience of the critical infrastructure and the solutions (or similar concepts) that are available in the literature. A careful evaluation of the state-of-the-art in other fields is advantageous, since the proposed architecture can easily borrow ideas and apply them on this field.
As network controllers move from their classical centralized model to highly distributed clustering software packages, understanding their internal architecture, programming models and performance metrics is also a central task before proposing an architecture.

Concerning the **development and implementation component** the goal of this thesis is to provide a framework/subsystem capable of:

1. Leveraging SDN to create logical sub-networks (basis for multi-tenancy).
2. Combining SDN and NFV (through container based virtualization) to create a virtual IDS service.
3. Exploring SDN and NFV as the technologies for the deployment of a virtual honeypot on the network.
4. Designing and implement a software emulated version of a data diode, via SDN and NFV
5. Exploring the architecture of the SDN controller to pipe relevant network events for further security analysis and correlation in a SIEM.
6. Targeting performance and availability (and security) above all other quality attributes.

The envisaged subsystem implies the development and implementation of an easily portable hardware testbed. Security probe deployments should be also made via a management web-interface (common to the whole ATENA project) to improve the user-experience and the likelihood of adopting the system in the IACS domain.

Proper architecture validation based on performance and availability metrics (as well as a set of real use cases for each of the developed virtual services) is also fundamental objective of this thesis.

1.4 Contributions

The research efforts of this thesis lead to the several outcomes listed below.

1.4.1 Research papers

– *Freitas, M. and Rosa, L. and Tiago Cruz and Simões, P. , "SDN-enabled virtual data diode", in Proc. of the 4th ESORICS Workshop On The Security Of Industrial Control Systems & Of Cyber-Physical Systems (CyberICPS 2018), September 2018*

The article is appended in Annex D of Volume II. Two more papers are currently being prepared for journal submission.

1.4.2 Research projects

Contributions to different research projects and their respective documentation/deliverables.

- **ATENA H2020 Project (H2020-DS-2015-1 Project 700581)**
  - Deliverable D4.1 - Requirements and reference architecture of the cyber-physical IDS (interim version)
  - Deliverable D4.2 - Distributed Intrusion and Anomaly Detection Strategies for IACS (interim version)
  - Deliverable D4.3 - Design of detection agents and security components (interim version)
  - Deliverable D4.4 - Design of the Distributed IDS for IACS (interim version)
  - Deliverable D4.5 - Requirements and reference architecture of the cyber-physical IDS (final version)
  - Deliverable D4.6 - Distributed Intrusion and Anomaly Detection Strategies for IACS (final version)
  - Deliverable D5.4 - Security Defined Software (interim report)
  - Deliverable D6.3 - Design and development report of the 1st release of components

  The following future deliverables will also include contents from this thesis:
  - Deliverable D5.9 - Security Defined Software (final report)
  - Deliverable D4.7 - Design of detection agents and security components (final version)
  - Deliverable D4.8 - Design of the Distributed IDS for IACS (final version)
  - Deliverable D7.4 - Final ATENA prototype validation and evaluation of validation results (Final report)
- **5GO P2020 - Mobilizador 5G (Components and Services for 5G Networks)**
  - Deliverable D2.1 - Use cases and requirements for solutions targetting 5G network core
- **Mobiwise P2020 SAICTPAC/0011/2015 Project**
  - The developed SDN subsystem is currently under viability evaluation for inclusion in the *Software Defined Orchestration* approach of the project.

1.4.3 Project reviews

Part of work performed in this thesis was of very importance to the success of the ATENA intermediate review in November 2017, at the European Commission. The implemented testbed was successfully replicated at the *Israel Electric Corporation* large scale testbed in May 2018, in preparation for the final ATENA review.

1.4.4 Open-Source contributions

Two pull-requests were made by the author and merged on the following upstream repositories:

- **OpenXENManager PR#132** - *Add support for multiple VNC windows* (B.Freitas 2017)
- **Open Network Operating System (ONOS) PR#18996** - *Remove host location when a switch port is removed* (B.Freitas 2018)

1.5 Structure of the document

The remainder of this document is organized as follows:

- Chapter 2 provides a short overview of the reference technologies used in this thesis. IACS basics, SDN and NFV concepts are explained.
  The OpenFlow protocol is described with more detail. The reader should consider this chapter as **optional** owing to briefly introduce the technologies.
  If the reader is already familiar with those technologies, this chapter can be skipped.
- Chapter 3 is dedicated to the state of the art. The evolution of IACS and the need for a paradigm shift are identified. A literature review on the use of SDN on IACS is provided, and SDN based architectures for security probe deployment are presented. Lastly, a review of container-based NFV approaches is provided.
- Chapter 4 is dedicated to software requirements. The contextual environment of this project (the ATENA IADS platform) is presented, along with its building blocks. The requirements elicitation process is explained and all the requirements (functional, non-functional and design constraints) are summarized in a uniform detailed way. Use case diagrams and descriptions are relegated to Annex A and Annex B (volume II) respectively.
- Chapter 5 is dedicated to software architecture. Distributed SDN controller architectures, clustering mechanisms, programming models and performance metrics are compared, in order to select the network controller for the platform (an architectural trade-off). Lastly, the architecture of the platform to be developed in this work is detailed.
- Chapter 6 discusses the adopted software development lifecycle and the whole work-plan. Components built from scratch in the context of this thesis (and component reutilization) are clearly identified. Development timelines, challenges and deviations

are explained. A reflexion is made concerning the proposed requirements (use case descriptions) and their completion.

- Chapter 7 is focused on the implementation and development component of this thesis. Network programming and the chosen algorithms are detailed as well as a brief reference to the external interfaces, application datastores and the management web-interface.
- Chapter 8 addresses the validation of the overall subsystem. Functionality is evaluated by defining a use-case per application. Non-functional validation targets the evaluation of the subsystem main design attributes: availability and performance.
- Chapter 9 concludes the thesis and provides some guidelines for future work.

Annexes to support the dissertation are provided in Volume II. They are organized as follows:

- Annex A contains the use-case diagrams organized by use case package. The overall system is stratified into functional blocks to improve the organization and readability.
- Annex B lists and details all elicited use-cases per functional package – identifying the success scenarios and respective exceptions.
- Annex C details the external interfaces of the developed subsystem organized per application.
- Annex D appends the published paper *"SDN-enabled virtual data diode"*. The paper provides a state-of-the-art review of commercial data diodes, explains the advantages and the challenges of virtualizing the data diode (bi-directional network requirements) and provides a proof-of-concept implementation based on the data diode application developed in this thesis.

Chapter 2

Reference Technologies

This chapter provides an overview of the reference technologies used in this thesis – it is of fundamental importance to familiarize the reader with their base concepts. Section 2.1 refers to *Industrial Automation and Control Systems* (IACS), identifying their types and main instrumentation. Section 2.2 discusses *Software Defined Networking* breaking the architecture into its three fundamental planes. Section 2.3 is dedicated to *Network Function Virtualization* (NFV) explaining how it complements SDN. It also provides a brief introduction to container-based virtualization. Section 2.4 discusses the *OpenFlow* protocol, the packet processing pipeline, flow tables and advanced protocol features. Finally, Section 2.5 provides a wrap-up of the chapter.

2.1 Industrial Automation and Control Systems

Industrial automation and control systems (IACS) are a broad class of command and control networks and systems used to support all types of industrial processes. They are the fundamental systems supporting critical infrastructures such as electricity generation transmission and distribution, gas production and distribution or water distribution. IACS includes a variety of system types, such as supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), process control systems (PCS) and other smaller control systems such as programmable logic controllers (PLCs) (MSec 2017).

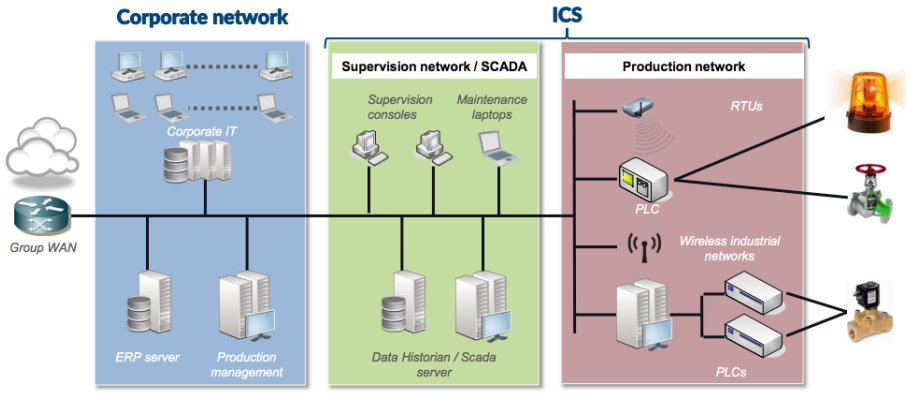


Figure 2.1: Generic representation of IACS (from MSec 2017).

Although normally referred as a synonym of IACS, SCADA systems (Figure 2.1) are in fact only a subset (a specific type of architecture) of industrial control systems. SCADA systems

are employed when centralized data acquisition is fundamental to the critical infrastructure operation. They reflect an highly distributed architecture used to control disperse process elements. Concrete examples include the oil and gas sector and power grids. In SCADA systems a "control centre" exists where centralized monitoring and control is performed. In the control centre operators (or an automated process) can push supervisory commands to field devices (to affect the process operation), can collect data from sensors, and monitor the distributed environment.

DCSs are mainly used to control industrial production processes such as oil, gas or chemicals plants. Contrarily to SCADA cannot be geographically distributed, but still entail a level of supervisory control which is extended to the whole production process. DCS typically integrates multiple control loops, each responsible for controlling one localized process section.

Despite the different subsets of IACS, some key elements are common to either SCADA and DCS (Endi et al. 2010):

- **Control Loop** - The control loop consists of sensors for measurement of process variables, controller hardware such as programmable logic controllers and actuators such as valves, switches and motors. The PLC interprets digital and analogue signals sent from sensors, computes deviations from the normal plant operation (set-points) and transmits control parameters to actuators. When a disturbance occurs in the process, signals sent by sensors to the PLC reflect the current state of the process making the PLC transmit new values to actuators so the process state can evolve as desired.
- **Human-Machine Interface (HMI)** - Graphical user interfaces that operators and engineers use to define set-points, control algorithms and to define the parameters of PLC's. The HMI also presents the process sensor values and historical information.
- **Remote Diagnostics and Maintenance Utilities** - Utilities used in IACS to prevent, identify and recover from failures

In current SCADA systems, control operations are stratified in three layers: supervisory control, process control and field instrumentation control (Figure 2.2).

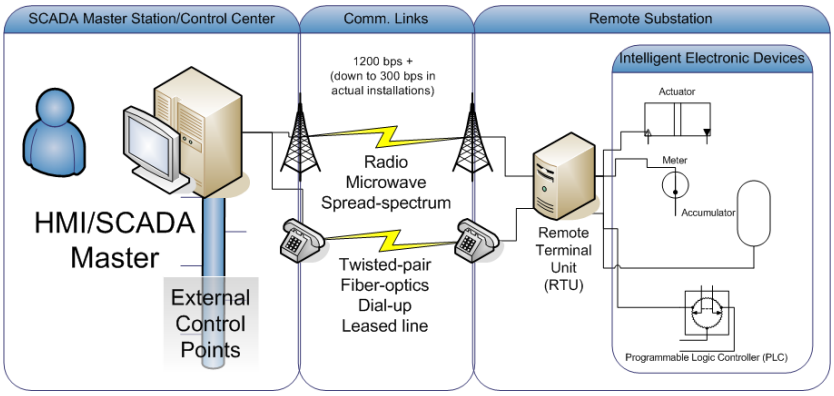


Figure 2.2: SCADA layers (from MachineryEquipment 2015).

The supervisory control layer (which includes master stations) corresponds to one or more general purpose computers which essentially have two functions:

1. Periodically obtain data from remote terminal units (RTU) and programmable logic controllers (PLCs) located in remote stations. This data is stored in historian servers which log process data over time.
2. Control remote field devices through commands sent to remote RTUs and PLCs.

Remote substations (the process control layer) are composed by several RTUs and/or PLCs. Traditionally, PLCs and RTUs had different implementation goals but, due to the increase of cheaper hardware, their provided functionality started to overlap (Endi et al. 2010). In the past, RTUs did not include control algorithms and were only used as wireless devices, providing single communication points to multiple PLCs. Nowadays, it is common to see both devices controlling the process field instrumentation layer interchangeably. Both equipments internally use *ladder-logic* to implement Boolean expressions – supported by switch or relay contacts (Ecmweb 2003).

The field instrumentation control layer (field devices) consists of sensors and actuators controlled directly by PLCs (and RTUs). Sensors get measurement data to feed PLCs, while actuators execute actions issued by the PLC. The PLC configuration and management is achieved in the SCADA upper layer – from the master station. Field devices are often *input-output* (I/O) devices used to signal the process status or to emit alarm signals. In some specific areas of IACS, like the electric power grid, field devices also include other microprocessor-based equipments such as: circuit breakers, transformers and capacitor banks. These devices are often categorized, and referred, as Intelligent Electronic Devices – IED (Techtarget 2017).

Regarding the communication infrastructure, data transfer within master stations uses generic ICT protocols such as TCP/IP or IPX over Ethernet (and Token Ring) (Thompson 2007). In the control layer, apart from Ethernet, other technologies such as RS-485, CAN, EtherCAT, Profinet or Industrial Ethernet are also used in what concerns the physical layer. Between master stations and RTUs/PLCs SCADA specific protocols, such as Modbus, IEC 60870-5-104 (IEC 104) or DNP3 are often used (Graveto 2017). For remote control, instructions are often sent through wide-area networks (WAN) using leased lines, radio or satellite technologies. The communication between the process control layer and field devices is typically point-to-point (wire pairs) – using voltage pulses or electric current intensity levels (4-20mA). Those values are interpreted by PLCs/RTUs depending on how they are programmed.

2.2 Software Defined Networking

Computer networks can be divided in three planes: the data plane, the control plane and the management plane. The data plane is represented by network equipment (routers and switches) that contain forwarding tables and are responsible for effectively forwarding data across the network. To efficiently forward data, forwarding tables in some network devices are called TCAMs (ternary content-addressable memory) since they can perform an entire table lookup in just one clock cycle. TCAMs greatly increase the speed of route-look-up, packet classification, packet forwarding and access control list commands. Such functions are responsibility of the data plane (TechTargetDef 2017). The control plane is the set of protocols used by the network device to populate the forwarding tables of the data plane elements. Finally, the management plane is composed of software services (e.g. SNMP

based tools) or web applications that enable network monitoring and management of the control plane.

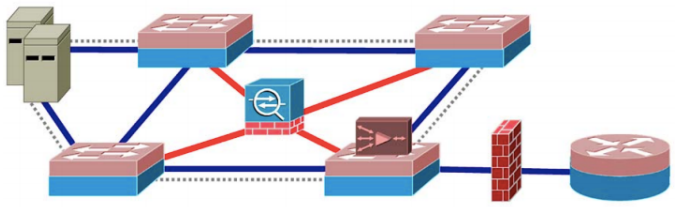


Figure 2.3: Tightly coupled control and data planes in traditional networks (reproduced from Kreutz et al. 2014).

Despite the exponential innovations seen in areas such as computing and storage virtualization, networking itself has remained essentially similar over the past 20 years. In traditional IP networks the control and data plane are coupled together in the same network device, and the whole management interface is highly decentralized (Figure 2.3).

The decentralized nature of traditional networks could easily be seen as an advantage since, hypothetically, it would lead to the improvement of the resilience of the network. However, managing such a decentralized infrastructure made clear some disadvantages that are more close to the architectural foundations of the network than to the amount of network elements:

- **Vendor lock-in**: The proliferation of network devices and the strategic choice of network vendors force customers to use products and services tied to a single vendor, which makes the transition to competitors' products hard to accomplish. Most of the time, to take full advantage of the network equipment, customers are forced to have an ecosystem where a single vendor provides the majority of software and services. This fact leads to a difficult transition to new protocols or architectures (Subramanian et al. 2016).
- **Management complexity**: Different equipment from different vendors have also different configuration instructions or interfaces. This results in a management nightmare, forcing network operators to have multiple management solutions and the corresponding specialized teams (Mousa et al. 2016).
- **Error prone**: Management of multiple devices, with different interfaces and instruction sets, is error-prone (Mousa et al. 2016). There is a big likelihood of producing inconsistencies in the network as a result of network configuration. This leads to poor network performance and even security holes (e.g. incorrectly configured firewall, wrong ACL entries, etc). Traditional ICT networks are also rather static and cannot dynamically respond to network attacks, load or faults.
- **Hampered innovation**: Capital and operational costs of managing the network are high with long return investment which hamper innovation and the addition of new features or services in the network (Sieber et al. 2016). Network equipment software (and hardware) is proprietary and brand-specific making research efforts difficult (and sometimes even illegal).

Software-defined networking is an architecture that attempts to solve most of the problems mentioned above by making the network programmable. The percursor of SDN can be considered the Active Networking, which appeared in the mid-1990s, motivated by the massification of the internet. Active Networking was an effort to open the network control plane through an envisioned programming interface that exposed resources (e.g. processing, storage and packet queues) on network nodes and supported the development of custom functionalities to subsets of packets traversing the network node. Later, between 2001 and 2007 efforts were made to separate the control plane and the forwarding plane from network devices. Examples of such efforts are ForCES, PCE and Ethane (Feamster et al. 2014). The fundamental development that made SDN gain sufficient traction was the introduction (circa 2007) of the OpenFlow protocol. One assumption of the early SDN research community was that network routers should be simple and homogeneous, working with IPv4 forwarding and Ethernet MAC switching. The OpenFlow protocol also assumed modern switches and routers contain TCAMs which could be exploited to construct flow tables in a simple match:action manner (Kreutz et al. 2014).

The idea behind SDN is quite simple: remove the control plane from network devices (leaving the forward plane) and shift it to a logical centralized remote location where network applications can define (and program) the network state (see Figure 2.4).

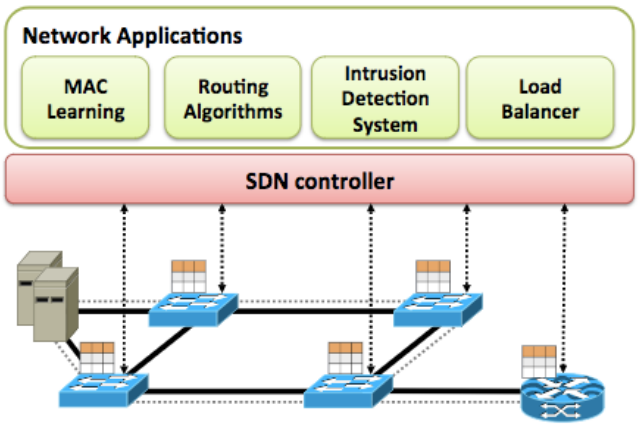


Figure 2.4: Software Defined Network Architecture (from Kreutz et al. 2014).

Recent developments in cloud computing and virtualization lead to the existence of multiple network programmability approaches. This fact has made the term SDN ambiguous since multiple authors refer to SDN as anything that is able to define the network behaviour through software. Nevertheless, to be considered SDN, the network architecture has to be supported by four main pillars as referred by Kreutz et al. 2014:

1. The control functionality is removed from the network devices, turning them into simple forwarding elements. In SDN, control and data planes are no longer tightly coupled.

2. SDN provides an homogeneous way of interfacing with the network equipment, making it possible to unify network behaviour and to virtualize the behaviour of traditional network middleboxes. The forwarding decisions are not based on destinations but instead on flow rules made of match fields and a set of actions (see Section 2.4).

3. The control plane is shifted to an external entity called the network controller or network operating system (NOS). The name comes from the fact that the features of the controller are similar to those of traditional operating systems although targeting network devices. The SDN controller is a software framework that runs on commodity hardware and provides the essential resources and abstractions to permit network equipment programmability from a logically centralized location.

4. The network is programmable by the means of software applications running on top of the controller, taking advantage of its essential services and abstractions. The network controller abstracts the communication protocols with the underlying network devices through a set of common interfaces exposed to network application.

Software defined networking segregates the network architecture into three main layers, as depicted in Figure 2.5.

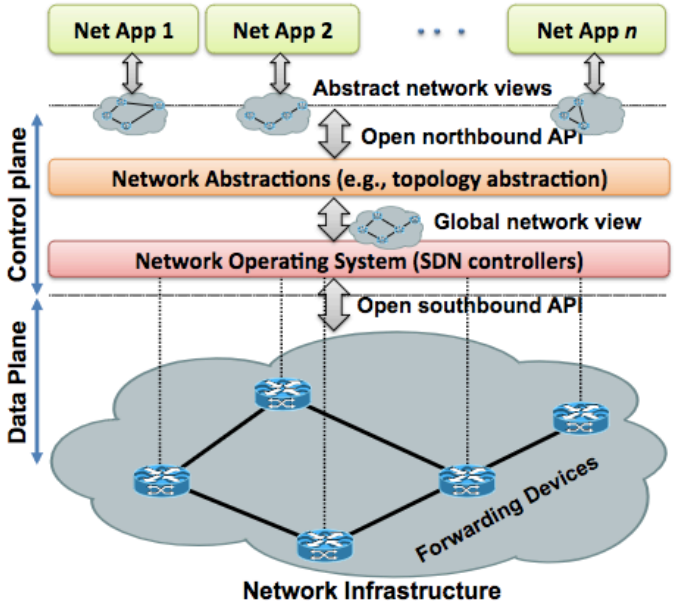


Figure 2.5: Software Defined Network layers (from Kreutz et al. 2014).

The separation of the network architecture through a set of different planes really entails the biggest advantages of SDN. SDN makes it easy to program network applications, since abstractions provided by the network controller core can be shared between applications. All SDN applications can access the same datastore, being aware of the global network state and topology. This fact makes network programming possible regardless of the location of the device. It is no longer needed to think about a specific strategy depending on the location of an asset (e.g. a middlebox) within the network topology (as opposed to what happens nowadays with traditional networking).

A bottom-up approach of the SDN architecture in terms of its respective planes (Figure 2.5) is depicted in the next subsections.

2.2.1 Data plane

The data plane in an SDN architecture is composed by the forwarding devices and by the southbound interfaces. Forwarding devices are provided by the SDN-enabled switch fabric (such as OpenFlow-enabled switches), composed by network equipment which usually do not have any forwarding decision implemented without being programmed by the network controller. These network devices have been called by some authors as "bare metal switches", in a clear allusion to bare metal hypervisors. Examples of such switches include those produced by Brocade, Pica8, Pantu or the Open vSwitch (an open source virtual switch implementation capable of being installed in any common physical host) (SDNCentral 2016). The southbound interface is a set of common Application Programing Interfaces (APIs) to abstract the communication protocols between the network controller and the forwarding devices. Albeit OpenFlow is the most common protocol, other protocols such as ForCes (which provides configuration for some non-openflow enabled switches), OVSDB (advanced management capabilities for Open vSwitch), OpFlex (an alternative to OpenFlow) or even SNMP can also be accessible through the southbound interface (Azevedo F. 2015).

2.2.2 Control plane

The control plane in the SDN architecture is composed by the following components:

- **Network hypervisors** - Network hypervisors are commodity virtualization servers where the network controller stack is installed. Hypervisors extend virtualization benefits to the SDN control plane, since they allow computational resource allocation in a shared pool of network controller nodes. Across SDN history, other projects such as Flowvisor (an OpenFlow proxy controller with the aim of creating virtual network slices) could also be part of this category (Azodolmolky 2013).
- **Network controller** - Software stack that provides abstractions (high-level programming interfaces) for concurrently access forwarding network devices and to program the network logic. The essential functionalities of the network controller include accessing the network state, device discovery, network topology, flow rule installation, device and flow statistics, and replication of the network state between multiple controller nodes. The controller is the critical piece of the SDN architecture and serves as the base for network applications to be developed. Multiple network controller softwares exist, from which OpenDaylight, ONOS, Floodlight and Ryu are the most known. The network controller can be centralized (and as a result be the master of all network switches) or can be distributed (sharing the network state and the switch mastership between multiple nodes). An overview of distributed controller architectures is provided in Section 5.1 emphasizing the architecture of both OpenDaylight and ONOS.
- **Northbound Interfaces** - Programming APIs that glue the control and the management plane together. There is little information on standardization of the northbound API. Some authors interpret the northbound API as only the external APIs other systems can access to interface with the controller (Banse et al. 2015). If that is the case, REST is without a doubt the most used interface. However, a few references (e.g. Kreutz et al. 2014) also consider the APIs between the main core services (e.g. topology, flows, etc) and their respective applications as northbound APIs, while these interfaces can be also viewed as internal to the controller (they are normally based

on remote procedure calls). Network applications can often extend the controller Representational State Transfer (REST) or command-line (CLI) APIs.

- **East/Westbound interfaces** - Interfaces responsible for horizontally connecting several controller nodes in a distributed control plane architecture. They are dependent on the controller software and are used to enforce synchronization algorithms such as RAFT or Gossip/Anti-Entropy (Naseer 2016).

It is also important to mention in this section the three ways the network controller can apply to define the forwarding plane logic if the OpenFlow protocol is used (NetworkStatic 2013):

1. **Reactive flow instantiation**: every time a new packet arrives into the switch a lookup is made by the switch in its flow tables. In the case where no match is found the packet is replicated, encapsulated and redirected to the network controller for it to take a decision. The network applications registered as packet processors/advisors are then responsible for forwarding or dropping the network packet. Additionally, they may install rules on the network devices to apply a similar logic to future identical network packets (either in a persistent or temporary way). The resulting overhead of this approach might by significant if we take into account the necessary logical steps and that not all network switches have flow tables built into specific application integrated circuits (ASICS) with Ternary content-addressable memory (TCAM) memory (Queiroz 2017).
2. **Proactive flow instantiation**: The network controller applications fill the network devices flow tables in anticipation, eliminating the need for the switch to redirect packets to the controller and the respective overhead of the process. This approach takes as an assumption that the network packets or the allowed communication between network hosts are known beforehand. It has a great impact on the system performance but compromises the system network flexibility.
3. **Hybrid flow instantiation**: Most of the flow rules are installed in advance by the network controller/applications ensuring low-latency to the network operation. However, if a packet that does not match any of the device flow rules reaches the network device it is forwarded to the network controller for it to take a reactive approach.

2.2.3 Management plane

The management plane is composed by the network applications responsible for receiving information from the southbound interfaces and for defining the way network packets are treated in the network. Applications make use of generic high-level object-oriented programming languages such as Java or Python (Trois et al. 2016) that abstract the network elements in the form of programming objects (device, flow or flow rule objects). During the evolution of SDN, virtualization languages with the goal of homogenizing the network programmability independently of the network controller were also suggested. One example is the Pyretic language (Pyretic 2015). However, the concept has not gained sufficient traction and those projects are no longer active.

2.3 Network Function Virtualization

Network Function Virtualization (NFV) offers new ways of designing, deploying and managing network services (SDxcentral 2016). It aims at decoupling the network functions from proprietary hardware appliances so they can run in software. More specifically, NFV makes use of virtualization technologies to virtualize common network functions (firewals, intrusion detection systems, DNS) so they can be consolidated in COTS hypervisors. The NFV concept can be traced back to the introduction of the first network functions: VLANS. However, its growth and continuous development can be attributed to telecommunication service providers. To accelerate the deployment of new network services, and recognizing the constraints of hardware-based appliances, telecommunication providers started to employ virtualization technologies in their networks. The European Telecommunications Standards Institute (ETSI) NFV group was formed to accelerate the research and to enforce common standards in NFV. More precisely, four working groups were created, defining the main NFV pillars (ONF-ESTI 2015):

- INF: Architecture for the virtualization Infrastructure.
- MANO: Management and orchestration.
- SWA: Software architecture.
- REL: Reliability and Availability, resilience and fault.

In September 2014 the Linux foundation announced the OPNFV platform as a means to provide a generic framework for the integration between SDN controllers and virtualization stacks (computing and storage) in open source ecosystems (OPNFV 2018). The platform heavily depends on OpenStack components for management and orchestration.

Network function virtualization (NFV) and Software defined networking (SDN) are complementary subjects. NFV can be used, for instance, for the virtualization of the network controller in the cloud, providing easy migration mechanisms. Another use, is to delegate network functions to virtual machines or containers in the SDN network. Software defined networking, in the other hand, complements NFV providing programmable network connectivity between virtualized network functions (vNFs), traffic steering to VNFs and network traffic optimization. Despite being complementary technologies, their concepts differ (Li et al. 2015):

- NFV is a concept for implementing network functions in software, while SDN is a concept for achieving a centralized programmable network architecture, to improve and control network connectivity.
- The goal of NFV is to reduce CapEx, OpEx and computational resource consumption accelerating the time-to-market of network solutions. SDN targets network abstraction to achieve flexible network control, configuration and a better environment for network innovation.
- NFV decouples network functions from proprietary hardware (easy deployment) while SDN decouples the control plane from the forwarding plane (easy network flow programmability).

To better understand how the two technologies differ (and complement each other), a software defined NFV system is shown in Figure 2.6.

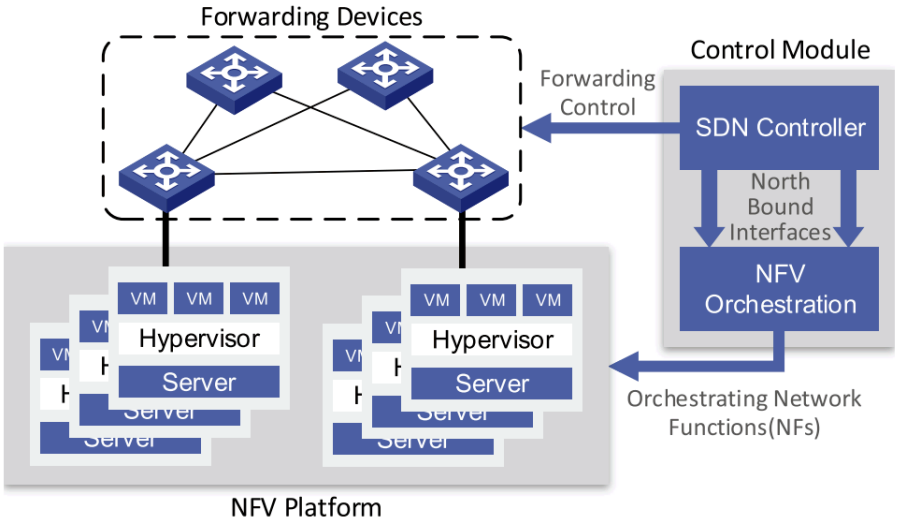


Figure 2.6: Illustrating SDN/NFV complementarity (from Li et al. 2015).

The SDN controller with an orchestration layer represents the logical control module of the system. The communication between both components happens through standard interfaces, i.e. the northbound interfaces of the controller. The orchestration layer can be a SDN application or an independent component which ties the external interfaces of the controller and the NFV platform interfaces together. The NFV platform is a set of hypervisors in which virtual network functions can be deployed in the form of virtual machines or containers. The control plane is responsible for orchestrating vNF creation, obtaining the global network topology and computing optimal paths (leading to vNFs) in the network. It finally programs network switches, using OpenFlow flow rules, to steer traffic to the appropriate vNFs.

2.3.1 Container-based virtualization

For a very long time, the term virtualization implied the use of *hypervisor-based* virtualization. An hypervisor is a piece of software that enables the execution of multiple operating systems (virtual machines) on the same physical hardware. They can be classified in *Type-I* hypervisors (if the hypervisor runs in bare-metal) or *Type-II* hypervisors (if they run on top of an existing operating system) (Combe et al. 2016). To simplify, the term hypervisor in this subsection is used to refer to *Type-I* hypervisors only. As virtual machines denote the exact same behaviour as real physical machines, hypervisors abstract the host physical hardware resources (CPU, memory, network) in a shared pool of logical resources from which each VM may draw. A mapping layer is created by the hypervisor to emulate the behaviour of real hardware. If we consider the virtualization of x86 platforms, this is not a trivial task. In the *hierarchical protection domains* model, Operating Systems are designed to run at ring 0 having instructions (and interrupts) that execute only in this ring. To virtualize the x86 platform, simply moving the OS to ring 1 without recompiling the OS force the hypervisor to perform binary translation which leads to a high performance overhead due to the use of unwanted emulation (Nagesh et al. 2017). Several advances have appeared in the virtualization field, mainly *Intel VT* (Intel Virtualization Technology) and *AMD-V* (AMD Virtualization) to enable hardware-based virtualization (also known as *paravirtualization*).

With *paravirtualization*, both the hypervisor and the Operating system share ring 0. The operating system calls the hypervisor services instead of using direct hardware resources (Marinescu et al. 2007).

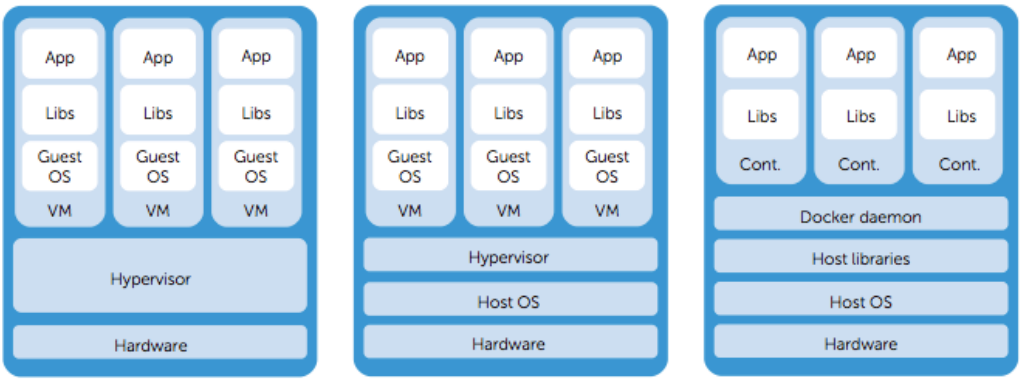


Figure 2.7: Type-I hypervisor (left), Type-II hypervisor (middle), Container-based virtualization with docker (right), (from Combe et al. 2016).

Despite the continuous improvements, hypervisor-based virtualization still brings a lot of overhead, specially considering the fact each virtual machine still represents independent and complete operating system stacks. On the other hand, *container-based* virtualization uses kernel features to create an isolated environment for running processes. Containers directly use the hardware of the host system without the need for emulating hardware or creating virtual device drivers. Several *container-based* virtualization solutions exist, depending on the host operating system: e.g. Solaris zones, BSD jails, LXC, OpenVZ, etc (Y. et al. 2017). In recent years, this kind of virtualization has gained sufficient traction. Specially Docker, a high-level API written in Golang around LXC containers, got mature and started to be highly used in production environments to sandbox specific applications (InfoWorld 2017). In fact, 79% of the Portworx Annual Container Adoption Survey 2017 (Portworx 2017) sample chose Docker as their primary container technology, showing that the Docker Engine project is still today a synonym of containers. For the sake of simplicity, containers will be mentiones throughout this document as LXC and specifically LXC managed by the Docker engine/API.

Figure 2.7 ilustrates the main differences in the computational stack of both *hypervisor-based virtualization* and *container-based* virtualization. In spite of the disadvantage of having to be compatible with the physical host CPU architecture, container-based virtualization has a huge performance gain since it completely strips the need for an hypervisor and uses features provided by the kernel for process isolation. Since they do not need to emulate hardware nor to boot a complete operating system, containers usually start in a few milliseconds and are far more efficient than classic VMs (Brikman 2017). Furthermore, container images do not need to contain a complete toolchain to run a full operating system. They are only composed by the binaries and shared libraries required to run the specific application which being isolated. Their small resource footprint, great scalability and security benefits are the main reason containers are becoming so popular.
The use of OpenFlow with linux containers requires some of the features LXC uses for process isolation. Hence, a simple analysis of some of these features is mandatory in the

context of this thesis. In the Linux operating system, LXC (which Docker inherently uses) take advantage of the following kernel features for process isolation (Eder et al. 2016):

- **Linux kernel namespaces**: Kernel namespaces are the foundation of process separation in Linux. Composed of several different namespaces (e.g. network namespaces), this feature allows the isolation of processes, groups of processes and even complete subsystems like inter-process communication or the network subsystem. This is the feature that enables containers to be paused or suspended, since each container processes are grouped in a different namespace. It is also possible to create a namespace which has a PID already used by the system or other containers, which greatly simplifies the container migration process. User namespaces are also possible through mapping of processes to user IDs, user groups and even other security-related identifiers such as the root directory.
  In the scope of container networking, linux namespaces (particularly network namespaces) make it possible to attach virtual network interfaces (*veth* - created on the host) to running containers by setting the network namespace of the *veth* to the PID of the container.
- **Control groups**: Although not completely required for process isolation, cgroups represent a mechanism to track processes (and process groups - including forked processes). They make possible the assignment of resources to each container and the further management of those assignments without unrestricted waste of physical resources. They also guarantee that physical resources are not unavailable when other processes claim the resources for themselves. This is the mechanism used in Linux to restrict resource utilization on a per-container basis.
- **Mandatory Access Control**: Mandatory access control is a set of mechanisms to improve security in Linux. Two of the most known MAC toolchains are respectively SELinux and AppArmor. Mandatory access control enforces polices when a resource is requested only allowing access if the policies are met. These security policies add another layer of protection to containers in order to mitigate attacks against the host (and other containers) from inside a container.

Another reason Docker is becoming so popular is the fact it facilitates container deployment and versioning. Part of this success is due to the way Docker handles container images and the high availability of template images stored in public repositories such as the Docker Hub.

Each container image is composed by a sequence of layers stacked on top of each other. When a container is created, a new writeable layer is created above all the other (underlying) layers. All the changes made to the running container like newly created or modified files are written to the top layer on the image layer stack (also known as *"the container layer"*). It is common for each container image to have a base template image (usually based on a specific linux distribution). Figure 2.8 shows a container running on top of an Ubuntu base image. Docker handles the interactions between image layers through a storage driver (e.g. Overlay driver) and uses a copy-on-write approach (Docker Documentation 2017).

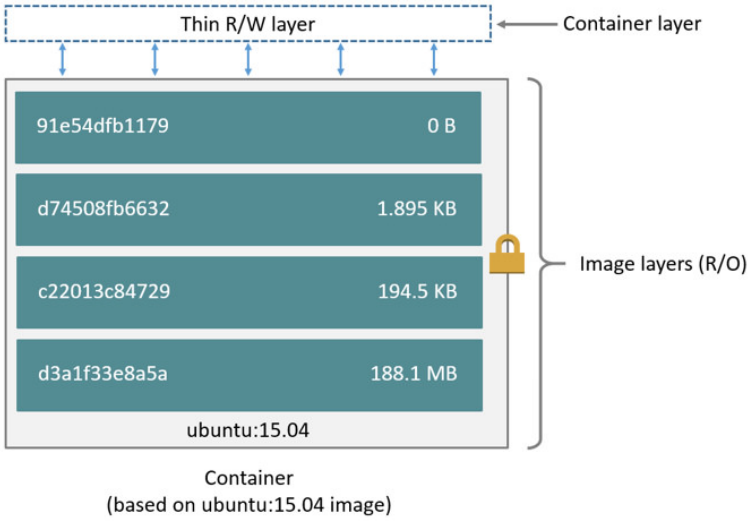


Figure 2.8: Docker image layers (from Docker Documentation 2017).

Regarding networking, Docker's networking model might itself be considered *Software Defined Networking* since container network behaviour is programmable through the Docker engine API. Docker provides several network options to containers (Docker Networking 2017):

- **Bridge mode**: Used by default if no other mode is specified, the bridge driver creates a private network internal to the host so containers can communicate with each other. Internally, the bridge driver creates the necessary linux bridges, virtual ethernet interfaces, iptables rules and host routes to enable the connectivity. The bridge mode automatically creates a `docker0` interface in the container being its IP address assigned by a built-in IPAM driver. In this mode, external access is only possible if port-forwarding is set on the container itself.
- **Host**: This network mode does not *"containerize the containers network"*. That is, all the hosts existing in the host network are also accessible to the container. This sort of network option raises, however, a few security concerns and severely limits the amount of services a host can run.
- **User created networks**: A few user-defined networking options were added to the latest versions of Docker; namely: `overlays` and `MACVLANS`. `Overlays` simplify many of the complexities in multi-host networking and are used by default in clustered (swarm) modes. IPAM, service discovery, multi-host connectivity, encryption, and load balancing are built in the overlay driver. With this mode, the user can create several logical networks between a set of containers. The `MACVLAN` driver, on the other hand, is a lightweight driver which directly connects host interfaces to container virtual interfaces avoiding bridging and port mapping. This approach creates a L2 segment from the container to the network gateway and as a result containers are addressed with routable IP addresses that are on the subnet of the external network (hicu.be 2016).

None of the options detailed above enables linux container networking to be defined by an SDN controller through the use of the OpenFlow protocol. This is a mandatory requirement in order to relegate network function virtualization to containers in the scope of an SDN controllable network. Connectivity (supported by OpenFlow defined flow-rules) has to exist

between containers and the other hosts on the network. Apart from this "issue", container-based virtualization still shows great promise as an NFV solution due to its lightweight nature and simplified image template model.

2.4 OpenFlow

The information presented in this section was obtained from the OpenFlow protocol version 1.5.1 (Open NF 2015), except when stated otherwise.

The OpenFlow protocol is a Layer 3 communication protocol that gives access to the forwarding plane of a network switch or router over the network. It enables network controllers to determine the path of network packets across the switch fabric. The original concept of OpenFlow originated at the Standford University in 2008 and the first version of the protocol standard was introduced in 2011 by the Open Network Foundation which has been managing the standard since then (Lara et al. 2013). As of today, Openflow is in version 1.5.1, although version 1.6.0 has been accessible to foundation members since September 2016. The protocol works on top of the Trasmission Control Protocol (TCP) although the communication between the controller and the switch can also make use of the Transport Layer Security (TLS). Controllers listen on TCP port 6653 for switches wanting to set-up a new connection.

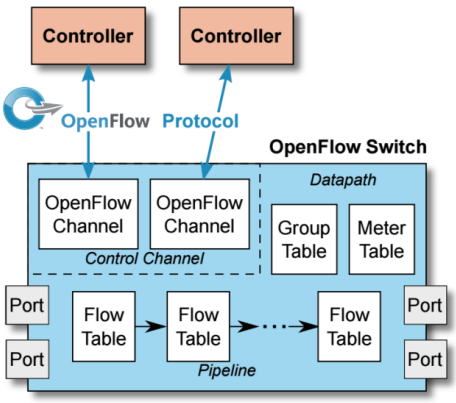


Figure 2.9: The OpenFlow switch (from Open NF 2015).

The architecture of an OpenFlow enabled switch is presented in Figure 2.9. The switch is composed by several secure OpenFlow channels (for controller communication), one or more flow tables, a group table and a meter table. When a packet arrives at a switch port, the switch checks their flow tables to find a flow rule matching the packet header. The corresponding action can be to drop the packet, forward it to a another flow table, forward it to a given switch port, flood the packet to all the ports or temporarily send the packet header to the controller and wait for a decision. Switches that support Openflow can be hybrid: some ports are controllable through OpenFlow while a few ports still behave as a regular switch. OpenFlow switches can be physical (e.g. datacenter solutions from Brocade, NEC, Cisco) or virtual – e.g. Open vSwitch (SDNCentral 2016).

The OpenFlow channel set up between the switch and the network controller is usually made over a secure connection (using VLANS). The switch can have a single OpenFlow

channel (being controlled by a single controller) or support multiple channels so the switch management is shared by multiple controller nodes. If the switch is configured to have multiple channels, one of the controller nodes is configured as the master of the switch. The remaining channels remain inactive and become active if the main channel fails. Hence, in case of a controller failure, another node will take the mastership of the switch.

The Openflow protocol defines three types of messages:

- **Controller-to-switch** - Connections initiated by the controller and used to directly manage the switch or inspect its state. These messages can be:
  - *Features* - Request the identify and the main capabilities of the switch.
  - *Configuration* - The controller queries and sets configuration parameters of the switch.
  - *Modify-State* - Used to modify the internal state of the switch. Examples of such messages are the insertion, removal or modification of the switch flow entries. Groups and action buckets are also configured through this class of messages.
  - *Read-State* - Messages used to collect information about the switch (e.g. flow statistics, port statistics, switch configurations).
  - *Packet-Out* - Messages used by the controller to send packets out of a specific switch port. The switch may be configured in a way any packet that reaches the switch and does not match any of the flows in the flow table is sent to the controller through Packet-in messages. Packet-out messages are the controller request to those messages.
  - *Barrier* - Used to receive notifications about the completion of switch operations. These messages are normally used to ensure message ordering.
  - *Role-Request* - Messages used by the controller to define the role of the switch. These messages are used to define the mastership of the switch.
  - *Asynchronous-Configuration* - Advanced messages used by the controller to configure the number and type of asynchronous messages it wants to receive from switches.
- **Asynchronous** - Asynchronous messages are sent by the switch without solicitation from the controller. They are used to inform the controller about a packet arriving the switch or to denote a state change on the switch.
  - *Packet-in* - Messages used to transfer the control of a packet to the SDN controller. Switches can support internal buffering and transfer only a part of the packet header (and a buffer id) to the controller. If the switch does not support buffering, the full packet is transferred to the network controller.
  - *Flow-removed* - Messages used to inform the controller that a flow rule was removed from the switch. The events leading to these messages can be the expiration of a rule (timeout is exceeded) or a flow removal request initiated by the controller.
  - *Port-Status* - Messages sent by the switch to inform the controller of a change in one of its OpenFlow ports. These events can be a change in the port configuration, a link that went down or a port that was removed by a user.

  - *Role-status* - When the mastership of a switch changes, the switch sends messages to the former controller master containing the new role-status.
  - *Controller-status* - Messages sent to all the controller nodes when the state of an OpenFlow channel changes.
  - *Flow-monitor* - Messages resulting from monitors set on the flow table. Once a change occurs in the flow table, the controller sends Flow-Monitor messages to the controller.

- **Symmetric** - Messages sent in either direction without solicitation. These can be acknowledgement messages, error messages or experimental message types. The OpenFlow protocol does not automatically guarantee message acknowledgement. Symmetric messages are used to keep the connection between the switch and the controller alive. As an example, the controller is allowed to ignore any message sent by the switch but it has to keep responding to symmetric messages to avoid closing the OpenFlow channel.

2.4.1 Flow tables and processing pipeline

One or more flow tables are at the heart of an OpenFlow switch. The processing pipeline (starting at flow table 0) defines how packets interact with those flow tables (Figure 2.10). When a network packet arrives at a switch, a table lookup is performed in the first flow table. The packet headers can then traverse a sequence of other flow tables if the switch is configured to have egress tables. Table lookups are performed in order to match packets against flow rules so a flow entry is selected. If one is found, the instruction set included in the flow entry is executed. If multiple flow tables exist, the action set can be configured to redirect the packet to a flow table with a higher number in the process pipeline (*Goto-table* instruction).

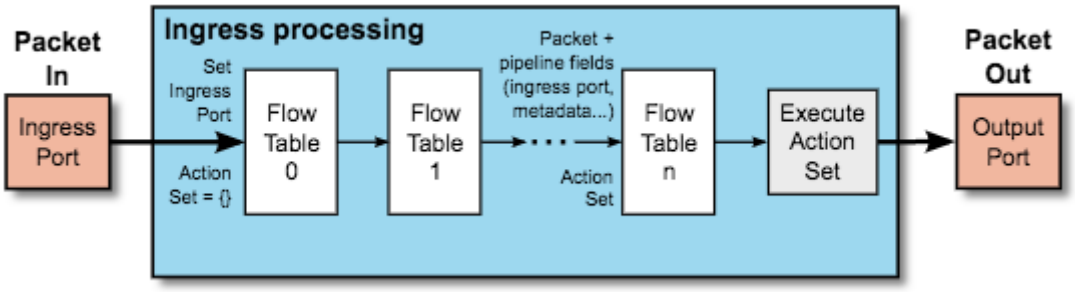


Figure 2.10: The OpenFlow processing pipeline (from Open NF 2015).

In case the switch only contains a single flow table, the processing pipeline ends with the execution of the actions specified by the matched flow rule. If the packet does not match any of the switch flow rules, a *table miss* occurs and the corresponding action depends on the table configuration. Usually, packets are sent to the controller in the form of *Packet-In* messages. However other actions such as dropping the packet or moving the packet further in the processing pipeline are also possible.

The structure of a flow table is represented on the Table 2.1. Details for each field are provided below.

Table 2.1: The OpenFlow enabled switch flow table.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

– **Match fields** - Ingress switch port and packet headers. Optionally, they might comprise other fields such as meta-data specified by a previous flow table. Used to match packets against flow rules, they comprise fields ranging from Layer 1 to Layer 4 and support both IPV4 and IPV6. Available match fields for OpenFlow are summarized in Table 2.2.

Table 2.2: OpenFlow match fields.

Match Field	Required	Description
IN_PORT	✓	Ingress Port (either physical or a switch defined logical port)
ETH_DST	✓	Ethernet destination address
ETH_SRC	✓	Ethernet source address
ETH_TYPE	✓	Ethertype of the packet payload
ETH_PROTO	✓	IPv4 or IPv6 protocol number
IPv4_SRC	✓	Source IP address
IPv4_DST	✓	Source IP address
IPv6_SRC	✓	Source IP address (IPv6 address format)
IPv6_DST	✓	Destination IP address (IPv6 address format)
TCP_SRC	✓	TCP Source port
TCP_DST	✓	TCP Destination port
UDP_SRC	✓	UDP Source port
UDP_DST	✓	UDP destination port

– **Priority** - Defines the priority order for each rule. Only a single rule can be triggered when a packet arrives at a switch port. Hence, priority is used as a selector if multiple rules apply to the same network packet.

– **Counters** - Properties that are updated once a packet matches a flow rule. The number of counter fields depend on the switch implementation. Not all the counters are required by the protocol specification. Counters provide usefull statistical information regarding each flow table, flow entry, port, queue, group, group bucket, meter and meter band. Counters also provide the duration a flow rule, port, group or queue has been available in the switch. The several types of counters are represented in Table 2.6. Please note that not all of them are required but some controllers (e.g. ONOS or OpenDaylight) implement many of the optional counters.

– **Instructions** - They define a set of operations to modify the usual packet processing. Examples of such instructions are `Apply-Actions` (apply rule actions immediately), `Clear-Actions` (remove all the actions if the rule is triggered), `Write-Actions` (append an action to the existing action set), etc. Once a packet matches a flow rule, the switch can then apply several actions as part of the instruction set. Those actions are summarized in Table 2.3.

Table 2.3: OpenFlow switch available actions.

Action	Required	Description
Output *port_no*	✓	The output action forwards the network packet to the specified OpenFlow port
Group *group_id*	✓	Delegate packet processing to a specific group (see 2.4.1)
Drop	✓	If no action or no group is defined the default action the switch will apply to the packet is to drop it
Set-Queue *queue_id*		Used in some switches as a way to provide QoS, this action forwards the packet to a given queue. Forwarding schedule depends on the queue configuration
Meter *meter_id*		Delegate packet processing to a meter SEE REF. As a result of metering the packet may be dropped.
Push-Tag/Pop-Tag *ethertype*		This action will push or pop specific ethernet tags from the packet. In case there are multiple actions, all actions are executed by the provided order. The associated data for this action may be – Push VLAN header – Pop VLAN header – Push MPLS header – Pop MPLS header – Push PBB header – Pop PBB header
Change-TTL *ttl*		Modify the time-to-live values for IPv4, IPv6 Hop Limit or MPLS.

- **Timeouts** - Maximum time (in seconds) until a flow rule expires and is removed from the switch.
- **Cookie** - String set by the controller. The cookie parameter can be used by the network controller to match several rules introduced by a single application (e.g. batch removal of flow rules).
- **Flags** - Flags alter the way flow rules are managed. For example, an `OFPFF_SEND_FLOW_REM` flag set on a rule will trigger the removal of the rule from the switch.

Group table

Group tables are the OpenFlow mechanism for applying a set of actions to multiple flow entries at the same time. The network administrator creates buckets of actions and associate them to a group identifier. Flow rule actions can then be created to move packet processing to the created group. The structure of the meter table and the definition of its parameters are presented below.

Table 2.4: The group table.

Group identifier	Group Type	Counters	Action Buckets

- **Group identifier** - Integer (32 bit) used to uniquely identify a group on the OpenFlow enabled switch.
- **Group type** - Used to specify the group behaviour. Actions applied in a bucket typically refer to packet modification (e.g. push-pop tags) or output via a given port. If the action group is empty, packets are dropped every time they are processed via the group. Group type semantics can be one of the following options:
  - *Indirect*: Only execute the actions of the first action bucket. As a result, packet processing is faster in this type of bucket.
  - *All*: Execute all buckets in the group. Each network packet is cloned before applying the actions of the bucket.
- **Counters** - Statistical information that are updated each time the group is triggered (see Table 2.6 for group available counters).
- **Action buckets** - This item contains an ordered list of action buckets. Action buckets represent a set of actions that are executed as a whole if the group is triggered.

Meter table

The meter table is the de-facto mechanism for rate-limiting (QoS) in the OpenFlow protocol. Metering in OpenFlow may consist of simple QoS modes (e.g. associating a network bandwidth to specific flow rules) or more complex modes based on DSCP (classifying packets in multiple categories based on their rates). The structure of the OpenFlow meter table is represented in Table 2.5.

Table 2.5: The meter table.

Meter identifier	Meter Bands	Counters

The main purpose of a meter is to measure the rate of packets assigned to it and, as a result, to control the rate packets exit the switch. They are a mechanism similar to queues although appended to flow rules instead of switch ports.

The parameters in Table 2.5 can be defined as:

- **Meter identifier** - Is a 32 bit integer that uniquely identifies the meter.
- **Meter bands** - Are a set of measurement bands that specify the rate of network flows and the way their packets are processed. A meter can have one or more meter bands but only a single band is applied for a flow at a time based on the measured packets rate. A flow which is mapped to a meter, directs packets to the meter which activates appropriate meter band if the measured rate of packets go beyond the rate defined in

meter band. The structure of a meter band can be found in Figure 2.11. Meter bands in OpenFlow have essentially two optional types:

- *Drop* - Discard packets if the rate limit is reached.
- *DSCP remark* - Change the drop precedence of the DSCP field in the IP header of the packet.

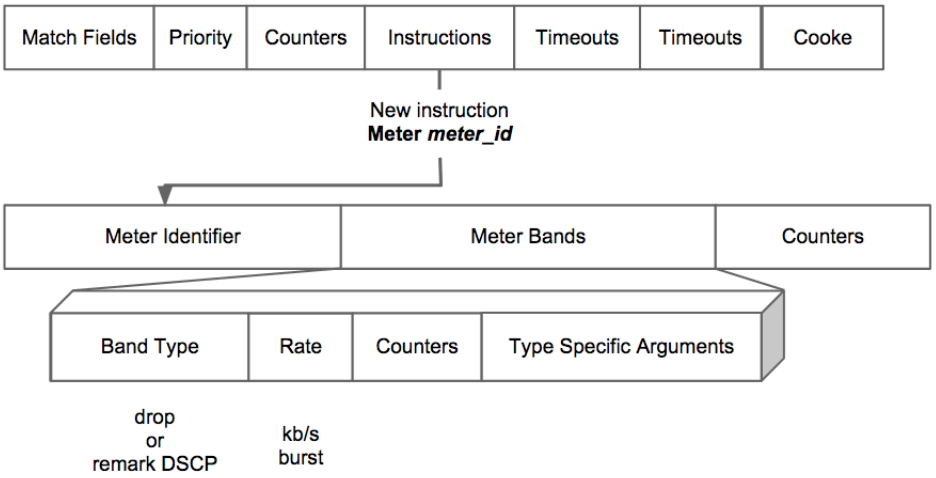


Figure 2.11: The role of meter bands in OpenFlow flow rules (from Raj Jain 2013).

In a meter band, the rate field is used by the meter to select the rate at which the band applies. Note that for each packet only a single meter band will process the packet. If multiple bands are defined, the first of the list is used.

- **Counters** - Statistical counters specific to meters. Those are updated each time a meter matches a packet. Counters are also available for each meter band (see Table 2.6 for the list of available counters).

2.5 Chapter wrap-up

Software defined networking defines a completely new paradigm making the network management process no longer a per-device task. Traditional networks limitations start to reveal once the number of interconnected devices scales. In the future, due to the advent of IoT, the global network size is only expected to increase with a broad new class of devices joining already existing networks. In traditional networking the configuration process is error-prone (creating possible attack vectors), use proprietary protocols, is complex on multi-vendor environments or force companies to adopt single equipment vendors hampering the potential for innovation in the field. By shifting the control plane from network devices to a centralized entity, SDN promotes global state awareness (even for complex networks) and makes it possible to have a new model: network programming. SDN applications are developed and installed in the network controller to program the underlying devices by the installation of flow rules. Network switches can also be instructed to send packets to the controller for additional inspection and processing. The importance of the network controller in SDN

is so relevant that they are also known by Network Operating Systems. Although several protocols exist to program the data plane, OpenFlow has emerged as the default protocol for SDN. It works in a match-action manner and has a fine grain control over each of the fields contributing to the overall flexibility of the architecture. It contains advanced mechanisms for quality-of-service assurance as well as device and flow rule statistics.

The *Network function virtualization* (NFV) concept adds a new layer of orchestration to the network controller, complementing the SDN approach. It approximates SDN implementations to the cloud computing model with the possibility of launching new virtual hosts on-demand. The SDN controller can request a new virtual deployment and then program the network flows to delegate packet processing to external machines. In this field, container based virtualization is gaining sufficient traction and is recognized to have better performance than virtual machines. Container based virtualization, specifically docker, was also detailed in this chapter.

Table 2.6: OpenFlow counters.

Counters	Bits	Required
Per flow table		
Reference Count (active entries)	32	✓
Packet Lookups	64	
Packet Matches	64	
Per Flow Entry		
Received Packets	64	
Received Bytes	64	
Duration (seconds)	32	✓
Duration (nanoseconds)	32	
Per Flow Port		
Received Packets	64	✓
Transmitted Packets	64	✓
Received Bytes	64	
Transmitted Bytes	64	
Received Drops	64	
Transmit Drops	64	
Receive Errors	64	
Transmitted Errors	64	
Received Frame Alignment Errors	64	
Received Overrun Errors	64	
Received CRC errors	64	
Collision	64	
Duration (seconds)	32	✓
Duration (nanoseconds)	32	
Per Queue		
Transmit packets	64	✓
Transmit bytes	64	
Transmit overrun errorrs	64	
Duration (seconds)	32	✓
Duration (nanoseconds)	32	
Per Group		
Reference count (flow entries)	32	
Packet Count	64	
Byte Count	64	
Duration (seconds)	32	✓
Duration (nanoseconds)	32	
Per Group Bucket		
Packet Count	64	
Byte Count	64	
Per Meter		
Flow count	32	
Input packet count	64	
Input byte count	64	
Duration (seconds)	32	✓
Duration (nanoseconds)	32	
Per meter band		
In band packet count	64	
In band byte count	64	

Chapter 3

State of the Art

This section provides an overview on the use of SDN in the IACS field. Section 3.1 details the evolution of IACS and explains the reasons why the current networking paradigm is in need of a complete shift. Section 3.2 presents some advantages of bringing SDN to IACS and details literature use cases. Section 3.4 is used to encompass all the security aspects related to the application of SDN in IACS. The section illustrates how SDN can serve to improve the security of the critical infrastructure and also details security vulnerabilities that might come from the shift to SDN. Moreover, literature proposals that adopt SDN-based IDs and honeypots are detailed in this section. Section 3.4 is dedicated to generic mechanisms for deploying SDN security probes. Section 3.5 explores NFV container-based virtualization and presents background work on how to use containers with SDN. Finally, section 3.6 provides a final discussion and explains what was learned from the literature review.

3.1 The evolution of IACS and the need for a paradigm shift

Today's industrial automation and control systems (IACS) are a result of a gradual evolution from analog wiring, towards digital lines, buses and finally networks. Since SCADA systems first appeared in 1960, the process control communication infrastructure was always intended to be mixed within the production line, and isolated from corporate networks. Control systems were designed as air-gapped "islands" where security was granted to the obscurity nature of their hardware and software (Eric Byres 2016). This fact traditionally was not perceived as a problem. At first, direct wiring between components allowed to rapidly detect any errors in the production line and to avoid their escalation to other parts of control network. Then, bus systems and serial solutions resulted as the natural evolution from analog wiring but have not brought substantial changes to the process control infrastructure: communication got digital but the main responsables were still electric technicians and engineers, instead of IT specialists.
In the 90's, business requirements with the goal of improving efficiency and productivity within process lines have forced IACS to get interconnected with corporate networks. In some cases, the geo-dispersed nature of the infrastructure have made them share wide-area networks and even be accessible through the internet (Pires et al. 2007). This generation of IACS broke with the isolated nature of the previous generations by including in its network designs open connections using TCP/IP. The Ethernet/IP protocol was developed while other common IACS protocols such as MODBUS and DNP3 were adapted to work on top of the TCP/IP stack (but not further enhanced to provide any sort of encryption (Drias et al. 2015)). A good example of a still undergoing modernization process is the power grid, which is transforming the electrical system into smart grids. Smart grids are

characterized by a two-way flow of electricity and information to create an automated and widely distributed energy delivery network that enables integration, effective cooperation, and information interchange among the many interconnected elements of the electric power grid (Panajotovic et al. 2011). In other words, the smart grid represents the perfect example to illustrate how critical infrastructures and their control systems evolved to adopt common ICT technologies. These connections allowed real-time monitoring, peer-to-peer communication from anywhere at any time, multiple sessions, concurrency and maintenance possible in IACS (Alcaraz et al. 2015). However, they came with the cost of losing isolation and greatly expanding the attack surface through the exploitation of vulnerabilities in the corporate network or in common ICT technologies and protocols. In the context of the power grid, with the advent of smart metering technologies, part of the critical infrastructure (CI) is also shifting to consumers' households, imposing several privacy and security concerns (Sadeghi et al. 2015).

It was not surprising that multiple cyber-attacks have arisen targeting the critical infrastructure and taking advantage of the inter-dependabilities between the cyber and physical domains. Stuxnet, a computer worm first uncovered in 2010, caused substantial damage to the Iran's nuclear program and made the world aware of the vulnerabilities of the once though safe SCADA architectures. While one could think such malware could only successfully infect the target due to some zero-day flaw in industrial control systems, the stuxnet worm (among other things) exploited a category of attacks well-known from the ICT domain: man-in-the middle attacks. The worm faked industrial process control sensor signals so the infected system did not shut down due to detected abnormal behaviour. The process centrifuges were gradually damaged while still looking under normal operation to process operators and engineers (Wired 2014).

In 2015, a cyber-attack against the Ukranian power grid, allegedly politically motivated, was able to successfully compromise information systems of three energy distribution companies in Ukraine and temporarily disrupt electricity supply to nearly a quarter-million end consumers (Kim Zetter 2015). Such an attack proved how a motivated (and resourceful) attacker can cause an outage on a country's essential services and defined the blueprint of cyber attacks that are yet to come. In 2016, it was found that cyberspies from China and Russia have hacked into the US electricity grid and hidden software that could be used to disrupt power supplies.

More recently, in 2017, TRITON, a new cyber-attack framework targeting IACS and with its roots in Stuxnet, was unveiled (Fireeye 2017). TRITON was designed to implement the TriStation protocol, a protocol used to configure Triconex SIS programmable logic controllers. The worm can send specific commands such as halt, read the PLC's memory contents and remotely reprogram them with an attacker-defined payload. Such attacks against critical infrastructures can have disruptive effects on essential services. This brings even more concerns if we take into consideration CIs are often inter-dependable with each other. For instance, a cyber attack targetting the power grid will also compromise the water distribution domain, since power is required in certain points of the infrastructure (Menashri et al. 2015).

The increasing number of attacks against critical infrastructures proves that IACS were not prepared for a connected world. This is mainly due to a common *"if it works don't change it"* mindset typically found in the industry. With the constant attempt to not compromise the usual plant operation (and the company's income) IACS tend to evolve rather slow, with small technological increments in terms of innovation. It is not unusual to still find hardware

and software dating from the 70's running for process control. Chemical industries, for instance, tend to use turn-key solutions (vendor lock-in) from reputable manufacturers and to outsource process control auditing (and maintenance) to third parties. What was once though as an intrisic security feature of IACS is now turning into a complexity bottleneck as the number of connected devices increases and IT engineers rush to keep the critical infrastructure safe from cyber attacks. In the future, as infrastructures evolve towards an IoT-generation of IACS, the number of connected devices and the overall network complexity is only expected to increase.

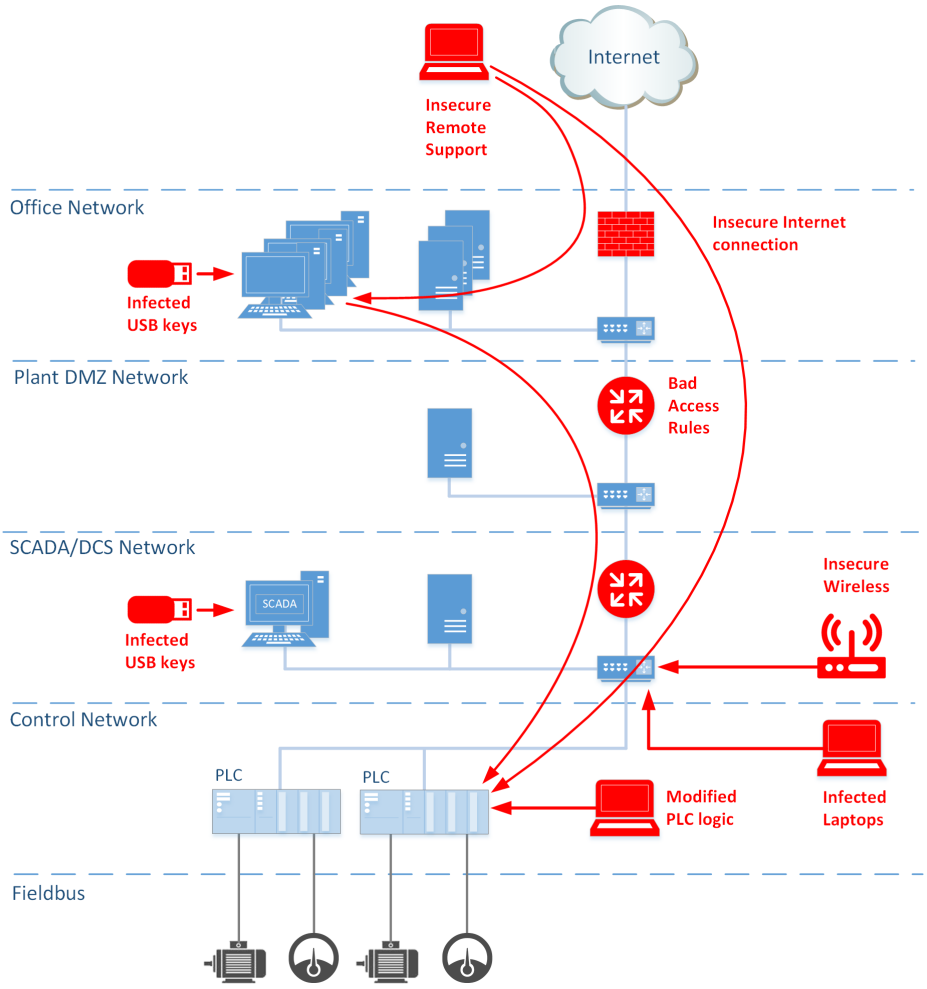


Figure 3.1: Network layers in IACS and possible attack vectors (from Kaspersky 2016).

In industrial Ethernet, unlike office networks and datacenters, the network is composed by a large number of small switches with low port count and only a few high-end switches (Gyorgy 2016). Furthermore, several network segments exist within the DCS network, normally formed through the partitioning of the broadcast domain at the data link layer (layer 2) by VLAN tagging and isolated with proprietary firewall middleboxes (Figure 3.1). The ISA99 committee is currently responsible for defining standards for implementing security practices and assessing electronic security performance in the IACS field (ISA99 2017). Every network device is configured and managed in a per-node basis using vendor-specific web interfaces or device-specific command line instructions. This leads to a very time consuming and

error-prone configuration process remarkably limiting the potential for innovation in the field (Mousa et al. 2016). Network reconfiguration or adjustment is also hard to be accomplished in the case of a cyber-attack. On the other hand, there is also the desire for network engineers working in IACS to have a global view of the network and to quickly identify any real-time misbehaviours. This goes in-line with what HMI devices nowadays provide for the control status of the production process. In traditional industrial ethernet networks monitoring is often delegated to the SNMP protocol (De Freitas 2012). Support for SNMP in network equipment, again, depends on the manufacturer of the device.

Although every industrial system has is own specificities, there are requirements that are usually common to all of them. One of them is network performance while providing redundancy at the link layer – thus not affecting the production process. As a result, it is common to find network topologies in the IACS domain not often found in other areas. Network ring topologies (e.g. Ethernet/IP) are a common way to ensure that all nodes in the network are dual-homed. The Rapid *Spanning Tree Protocol* is used to disable redundant links and to keep the quality of service level in case of a link failure. This kind of protocols may, however, not be suitable to keep the network performance as industrial networks scale (or even in some IACS applications like the power grid due to strict time-recovering constraints) (Minicz et al. 2017).

Hence, IACS are in urge of simpler and more systematic network designs. To keep up with the scalability challenges while keeping performance, QoS and manageability, newer network approaches should be considered and evaluated for IACS. Preferably, network solutions that aim to simplify the data path layer and accommodate the differences introduced by each vendor in the network equipment, since the majority of IACS traffic focuses on the Layer 2 level (Gyorgy 2016).

3.2 SDN in the IACS domain: benefits and use-cases

At the same time network operators struggled to manage traditional networks, a new network architecture originated around 2009 from the academia: *Software Defined Networking*. Software defined networking (see Subsection 2.2) removes the control plane from forwarding devices and transfers it to a logically centralized location called the network controller. The network controller (also known as Network Operating System) keeps a global view of the network and relies on specially crafted network applications to flexibly and dynamically manage forwarding devices. Combined with the Openflow protocol (c.f. Section 2.4) network switches are turned into *"dumb devices"* only meant to be controllable from a central place through the instantiation of flow rules. It was only a matter of time until the advantages of SDN and Openflow started to gain acceptance in production environments, pushed by internet giants such as Google, Facebook and Amazon, and telecom companies from which AT&T and Deutsche Telekom are the best examples (Cox et al. 2017). OpenFlow set the precedent for a stardardized vendor-independent interface between a centralized control-plane schema and a number of distributed data-plane entities. Leveraging on OpenFlow control-plane architectures, large cloud providers began to develop proprietary Software Defined Networks (SDNs) to create production-quality, global-scale, single-tenant control plane architectures, e.g. Google's internal WAN and NTT's Enterprise Cloud (Argyropoulos et al. 2015). Cloud computing revolutionized the way networking is managed. At a distance of a single click new networks are deployed/built following a multi-tenancy environment and centralized management (Wang et al. 2017).

As defined by the Open Networking foundation (O.N.F. 2012), SDN entails a number of advantages when compared to traditional networking. Hence, academia was also quick to address the advantages of SDN in the context of IACS scenarios.

Although mostly focused on the power-grid use case, Da Silva et al. 2015 showed how the *fault*, *configuration*, *accounting*, *performance* and *security* of traditional SCADA systems can be greatly improved by taking advantage of certain characteristics of the SDN architecture, namely:

- **Programmability**: allowing the creation of customized services in the SCADA network (e.g. applications to control the reading frequency of field devices).
- **Flexibility**: easing the addition of new field devices and the upgrade of existing network applications in the SCADA network.
- **Centralized management**: promoting the creation of a SDN-SCADA control center which allows not only the management of field devices but also to monitor and control the network which interconnects them.
- **Standard API**: through the use of open standards such as the OpenFlow protocol geographically dispersed multi-vendor SCADA equipment can be better integrated.

Sharma et al. 2016 share the same vision. They further identify that SDN can contribute to green networking, since most of the heat generated by network equipment is due to computational effort of their control plane. SDN decouples the control plane from the data plane, delegating the first to the network controller who treats the network as a whole and thus reducing the overall energy consumption of forwarding devices.

Kim et al. 2015 also identify several advantages of bringing SDN to the grid network:

- **Interoperability** - By using the OpenFlow protocol as the default standard, SDN can guarantee fine-grained control over all the network devices in the IACS infrastructure. Service providers can access and control the device behaviour without the isolated stack regulations of each device.
- **Situational awareness** - With SDN, the service provider can monitor the real-time network status by using information obtained from each device. Furthermore, this information can be used to implement custom logic in SDN applications to quickly adapt to the current network state. In the current grid architecture, the service provider may not understand the need of upgrading the infrastruture stability without further network equipment (firewalls or other packet inspection systems).
- **Simplified service deployment** - The network controller on a SDN smart grid would be located in the control center where service providers could deploy flow rules without the need for new hardware installation and configuration. With SDN, the service application is embedded within the application layer of the SDN controller.
- **Simplified business model**: The SDN-enabled smart grid business model could be simplified to include policy development, service provisioning and service monitoring. In this business model, the system administrator just deploys an SDN application, such as topology construction, routing, QoS and traffic-filtering program to enforce a given profile.

Sainz 2017, albeit pointing other potential advantages, states the core change SDN can bring to IADS is due to securing the critical infrastructure. Security opportunities and

the respective challenges of SDN-based IACS are quite a broad subject. Although many published works detailed in this section are also inherently bringing security since they focus on improving the infrastructure resilience, a full subsection is dedicated to the security implications of SDN-IACS (Subsection 3.3).

Several studies and use-cases have been proposed to evaluate how *Software Defined Network* could change the current status-quo of IACS networking, some are presented in the next text.

Goodney et al. 2013 propose to use SDN for data synchronization between multi-station synchronous *Phasor Measurement Units* (PMUs). PMUs are used for wide area measurements and control of the grid network. They operate by sampling the state of a power line and by streaming data to a single client. In order to guarantee a global view of the grid state, multiple clients are interested in receiving the data stream (thus following a mechanism similar to the *publish-subscribe* software pattern). So, the grid contemplates the existence of PDCs (phasor data concentrators) which concentrate data from multiple PMUs and sends it to clients (often dispersed across regions). In PMU networking, IP multicasting solves part of the problem: clients either join an existing multicast group or create a new one. However, according to the authors, this approach introduces a few problems: PMUs would have to be redesigned to transmit data in the multicast format, support multiple group addresses and support UDP checksum calculations for data integrity. Also, with multicast trees the problems of network contention are also important since multiple copies of the same packet are sent over the same link. Moreover, network optimization is poor and some links have to support high throughput to compensate the clients' bitrate needs. SDN, on the other hand, flattens the network topology. It overcomes the need for PDCs since flow rules are created to provide copies of the traffic to all the subscriber switch ports. Network packets are replicated at edge devices, which significantly lowers the load on certain network links and the network latency optimizing the overall network. Figure 3.2 shows the advantages of SDN when compared to traditional IP multicasting for PMU networking.

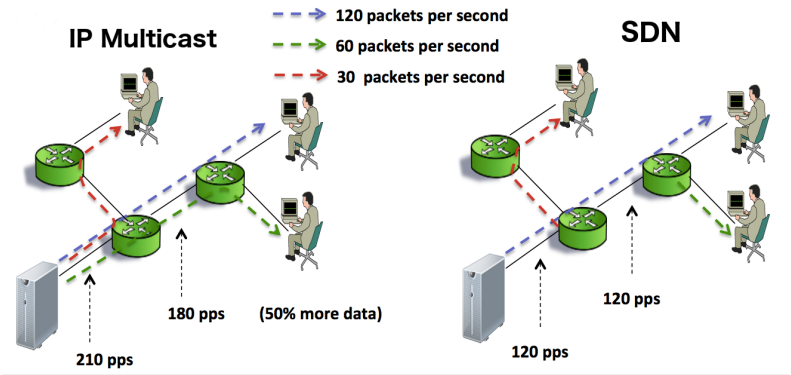


Figure 3.2: Advantages of SDN over IP multicasting for PMU networking (adapted from Goodney et al. 2013).

Cahn et al. 2013 demonstrate how NFV can be coupled with SDN to take advantage of modern computation approaches such as those of virtualization and *Fog computing* in the energy communication network (SDECN). The authors analyze the recent developments

of intelligent electronic devices (smart meters) which are evolving to simple devices that *"packetize"* measured values over *Ethernet*. So, this would allow the creation of a power grid where multiple measurement devices delegate data processing to virtual machines close to the sensor location whereas SDN would be the means to achieve network connectivity between them. Furthermore, the authors also emphasize how SDN could be used to create virtual (hence logically separated) sub-networks in the power grid leading to a *multi-tenant* environment. In SDECN, network tenants would be seen as each virtual sub-station that could be dedicated to a particular costumer, a utility, an energy source or even a region. The authors also provide an analogy between *IaaS* cloud computing model and the hypothetical *Grid-IaaS* model.

A similar approach is also followed by Cruz et al. 2016, although applied to a different subset of IACS equiment – *programmable logic controllers* (PLCs) – aiming towards a *vPLC* concept. A great level of detail is dedicated to the viability of such a solution by evaluating details often neglected by similar proposals. In fact, the virtualization of industrial grade equipment has a different set of requirements when compared to general-purpose workload virtualization in COTS hypervisors. It requires real-time operating systems (RTOS) in which low latency and determinism are essential. x86 virtualization in COTS hypervisors often prioritize throughput using techniques such as resource sharing, deferred interrupt processing, *hyperthreading* and frequency-scaling which impact the latency of real-time (RT) applications. However, the authors present how some of the developments in the x86 virtualization domain (e.g. system management interrupts) may also favour real-time virtualization and also point out the presence of the first real time hypervisors. Thus, a solution for a virtual PLC is presented which replaces the traditional I/O bus by a deterministic and a high-speed network infrastructure supported by SDN (Figure 3.3). SDN is then proposed as the vehicle which enables connectivity between the vPLC and their I/O physical modules (implemented using Field-programmable gate arrays (FPGAs) or common integrated circuit (IC) components). The RT hypervisors integrate well with other IACS components already being virtualized nowadays (e.g. human machine interfaces - HMI) in the proposed solution.

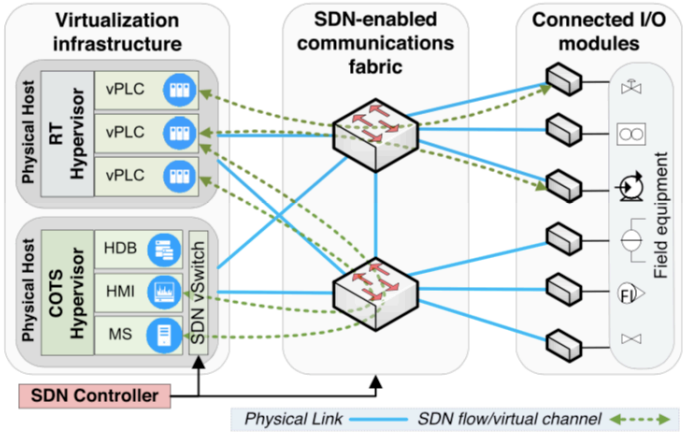


Figure 3.3: Towards the virtual PLC. (Cruz et al. 2016)

Another important aspect presented by Cruz et al. 2016 is the enhancement of the SDN network through the use of Intel's *Data Path Development Kit* (DPDK). DPDK enables low latency and high-throughput packet processing (on supported hardware) bypassing the operating system kernel and bringing the network stack directly to the user space. This

severely reduces packet processing overhead leading to network applications able to perform direct memory access (DMA) operations.

In the scope of the Smart Grid IACS, much attention has been devoted to the IEC 61850 standard and the need to guarantee near real-time network adjustment in the occurrence of a topology link failure. This standard recommends delays from 3 to 100ms for protection messages depending on the message type. Stricter standards such as the IEC 1646 set the maximum delay requirements as little as 4 and 5ms for 60Hz and 50Hz AC frequencies. These time thresholds are, however, smaller for applications relying on the communication between the power grid sub-stations. The remote activation of a protection scheme at a sub-station must occur within 8-10ms after the fault has been detected. As a result, multiple authors have been evaluating SDN as a way to meet the IEC 61850 standard while the network scales.

Molina et al. 2015 proposed a smart grid SDN management application for an IEC 61850-based smart grid system. The system presented an algorithm capable of translating a sub-station configuration description (SCD) into OpenFlow flow rules. The SCD is an application profile of the IEC 61850 standard and indicates various metering profiles, such as monitoring intervals, attributes and types. The authors were capable of using the SDN controller real-time flow monitoring features to detect potential denial of service attacks without the need for additional network monitoring devices such as firewalls. The authors also stand out the aptitude of the SDN controller (Floodlight) to enable routing, traffic filtering, QoS and load balancing in the smart grid network.

Aydeger et al. 2016 showed how SDN can be applied in Smart Grid communications to provide redundancy between sub-stations. The authors forced two hosts (sub-stations) with multiple network interfaces to share TCP MMS (manufacturing message specification) packets each 4ms and then forced the removal of an existing wired link. The change in the global topology was detected by the network controller (OpenDaylight) which adapted the flow rules to use the backup link. Thus, the authors conclude the SDN architecture can be easily used to provide redundancy within the smart grid network despite not providing any performance test metrics.

ARES (Lopes et al. 2017) is a recent SDN-SCADA based architecture suited to meet the IEC 61850 standard requirements. In this study, the authors compared the recovery times of SDN with those achieved through the use of traditional fail-over protocols such as the *Rapid Spanning Tree Protocol* (RSTP). RTSP typically has recovery times in the order of a few seconds. On the other hand, ARES combined with the OpenFlow protocol version 1.3 was able to achieve average recovery times of 0.6 ms. In order to reach such small recovery times, ARES used a proactive flow instantiation approach (setting layer 2 multicast trees in order to reduce the impact of layer-2 flooding). Interestingly, the authors also add another application layer (called SCADA-NG) above the controller core which enclosed SCADA applications requiring automatic interaction with the core network. Monitoring of the SCADA smart grid assets such as DERs, EVs and smart meters is also responsibility of the SCADA-NG layer. The overall architecture is presented in Figure 3.4. Please note that despite the small recovery times achieved by the ARES proposal, the study is merely conceptual and the selected SDN controller (Ryu) is not a distributed controller. Hence, it easily represents a single point of failure within the smart grid.

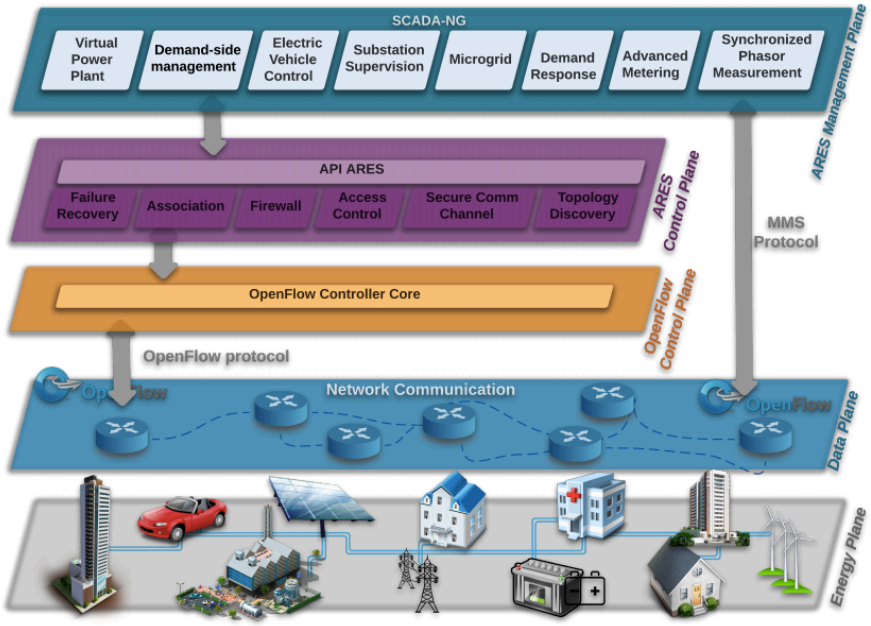


Figure 3.4: ARES architecture (Lopes et al. 2017).

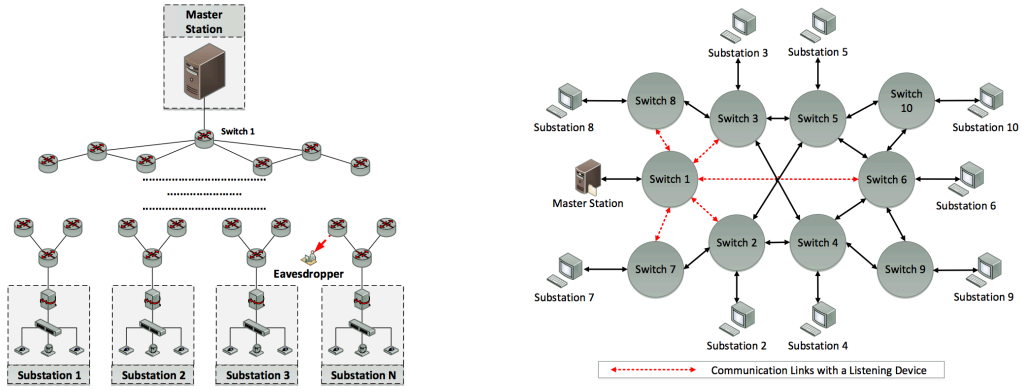
Either way, the efforts in studying and evaluating SDN in the IACS domain lead us to the point where the first SDN turn-key commercial solutions are finally (although shyly) hitting the market. In 2016, *Scheitzer Engineering Laboratories* (SEL), a company well-known in the automation field for their sub-station automation equipment applied in power production and distribution, launched an Openflow-based flow controller and a set of configurations for some models of their industrial Ethernet switches (Scheitzer 2016). The SEL suite targets SCADA sub-stations which typically are composed of a small number of message types and redundant links. The company's flow controller is able to both assign rules to industrial switches and enforce path redundancies in the network and to forward all non pre-defined packets to the network controller. This succeeding aspect is pointed by sub-station engineers as the most relevant since they now can become aware of any packets entering the network which do not match the expected network behaviour (Forbes 2017).

Another example is the recent announcement from the *Yogagawa Electric* group stating the company is using an OpenFlow system in four paper mill plants owned by the *Oji Holdings Corporation* (Automation.com 2017). In this case, SDN is being used to secure the enterprise-to-plant sub-network routing. As seen before, the common industry practice is to isolate both networks recurring to the extensive use of firewalls and other security devices. These appliances are limited and cause difficulties if legitimate enterprise applications (for supervision or support of the manufacturing operations) need to run at the enterprise network level.

3.3 SDN in the IACS domain: security aspects

IACS architectures, as seen in Subsection 3.1, are insecure by design as they were never developed to be connected to external systems. Despite the existence of some encrypted extensions to industrial control protocols, most of IACS systems still rely in unencrypted message exchange. Priorities for IACS and ICT are opposites: for IACS availability comes

first even if at the cost of losing integrity and confidentiality. Packet eavesdropping which can easily lead to replay attacks in the network are a severe problem in SCADA systems (Mo et al. 2012). Da Silva et al. 2015 propose to address this issue with SDN by adopting a multi-path routing strategy. They took advantage of the *Timeout* field of the OpenFlow flow table to implement an SDN application capable of dynamically computing the available paths between two hosts in the network and to install an expiring flow rule matching one of the computed paths. To be able to choose a different flow path, the application maintained a local path storage and registered itself in the controller as a *Packet Processor*, i.e., an application capable of receiving and react to *Packet-in OpenFlow* messages sent by the network controller. The proposed algorithm was able to greatly reduce the percentage of intercepted packets exchanged between the master station and the grid sub-stations (by a factor between 25-75% - Figure 3.5).



(a) Multipath test bed implementation

(b) Available routes from the master station to each SCADA sub-station

Figure 3.5: *Multipath routing* based approach to mitigate *eavesdropping* in SDN-SCADA networks – (reproduced from Da Silva et al. 2015).

Additionally, Dong et al. 2015 adopt a moving defence strategy by proposing to use SDN to establish dynamic routes for the grid control commands. Dynamic routing would prohibit malicious re-routing and denial-of-service (DoS) attacks. Moreover, the authors propose to use SDN to reset switches and to re-establish routing upon the detection of a compromised device. A reactive architecture was developed introducing a *"control center"* composed by the cooperation and data synchronization between a SCADA IDS, the SCADA master and the SDN controller. Possible attacks against the new network architecture could be categorized into three main classes:

1. Compromised network switches.
2. Compromised grid devices (such as SCADA slaves, RTUs and relays).
3. Compromised SDN controllers and their applications.

Their paper illustrates how the SDN controller can be used to defeat attacks 1 and 2 by establishing a route to transmit control commands only when necessary, shortening the time window an attacker can use to inject malicious commands from a compromised switch or grid element. Such mechanism of blocking network traffic in certain link directions is well-known within the IACS domain and is typical of unidirectional gateways (Heo et al. 2016). SDN can also be effectively used against DDoS attacks (i.e. if a compromised grid asset spoofs packets that request sensors or relays to send measurement data to a specific RTU or

data aggregator) since the network controller can easily adjust the QoS of a certain network link or route in the network topology. The effectiveness of SDN to improve QoS routing in smart grids is also detailed in J. Zhao et al. 2016 in which the shortest path between network hosts is computed taking into account parameters such as the link throughput. In case a significant section of the grid is unavailable, Dong et al. 2015 state SDN provides the perfect environment for hot-swapping between private and public networks, borrowing resources from cloud providers through secure channels. Furthermore, the SDN possibility of creating logical networks within a global network also contributes to the mitigation of such high grid damage. In what concerns the compromise of the SDN controllers, the authors propose to use the available *"control center"* IDS to inspect every single OpenFlow message sent and received by the SDN controller. A testbed was build using the NOX SDN controller and Bro network security monitor as the IDS.

The above examples show how network intrusion detection systems play an essential role in securing industrial control systems. In fact, current automation security practices recommend that the network should be partitioned and each security shell should comprise a network IDS (Mckay 2012). SCADA traffic is somehow different from those found in other environments. Flows are periodic and the connection matrix is rather static (Barbosa 2014). Datasets concerning SCADA vulnerabilities and attacks are scarce, since industrial companies do not want to expose such data to avoid future attacks (Cagalaban et al. 2011). As a result, classification algorithms are often used and reported to have high levels of attack detection in SCADA networks. A good example is the work published by Cheung et al. 2006 which used three different models to detect anomalies in MODBUS TCP network traffic. Multiple other authors have been proposing SDN IACS architectures that contemplate at least a network IDS.

E. G. Da Silva et al. 2016 propose a SDN-Based SCADA IDS inspired on *big-data* concepts (c.f. Figure 3.6). The authors extend the Historian Server typically found in traditional SCADA systems to also store snapshots of the network packets flowing in the network.

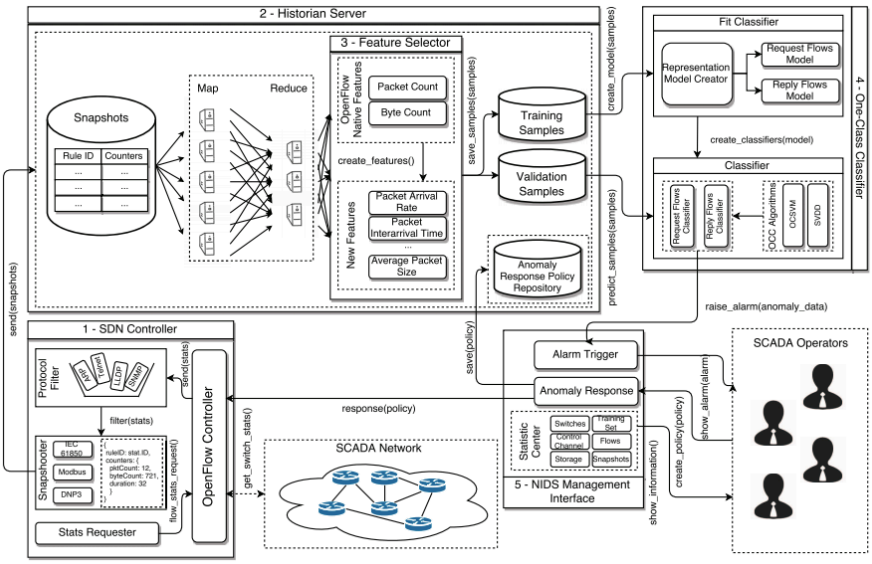


Figure 3.6: SCADA IDS architecture sugestted by (E. G. Da Silva et al. 2016).

Map-reduce in a distributed fashion is used in the historian server to find flows matching a given packet header and to reduce them into searchable keys. These keys are used to train a one-class classification machine learning algorithm based on support vector machines (SVM). The classifier continuously evaluates snapshots of the network traffic predicting the probabilities of being representative of an attack. In that case, it generates an alert which is visible in a management interface which SCADA operators can use to take further action. In the provided concept, SCADA operators would define network policies when an attack is detected that are then translated by the network controller in a set of flow rules through the OpenFlow protocol. Examples of such actions are identified by the authors as blocking the traffic or redirecting the attacker to a HoneyPot. The published work claims to be able to detect 98% of the DoS attacks committed against the infrastructure.

Lallo et al. 2017 also recognize the importance of using a network IDS to monitor the critical infrastructure. By applying a non-automatic approach (i.e. delegating decisions to the infrastructure operators) the authors state nor all of the CI traffic has the same monitoring priority. Hence, they propose to leverage SDN in combination with an integer linear programming (ILP) solver to find which flows should be copied to an IDS. CI operators firstly identify the most relevant network equipment and protocols and the solver is run aside from the network monitoring process (to avoid the creation of a bottleneck).

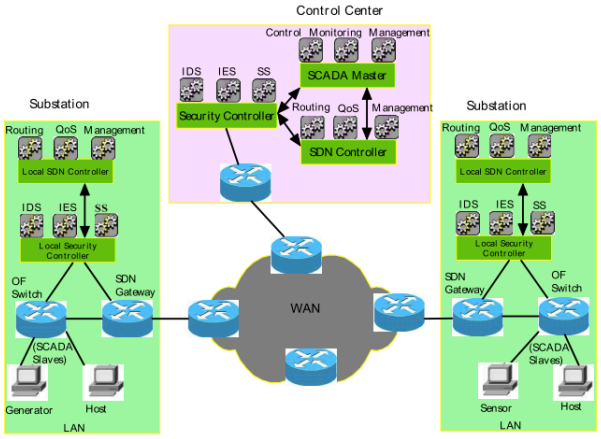


Figure 3.7: The security framework for SDN-enabled smart power grids (Ghosh et al. 2017).

Ghosh et al. 2017 raise an interesting point: most of the available SDN-SCADA research focuses on protecting the infrastructure against outsider attacks while only providing security assurance within the cyber (and SDN) domain. Insider attackers that may influence the power grid as a whole are often overlooked. Hence, the authors propose an architecture composed of several network controllers and multiple intrusion detection systems. A local IDS is deployed in a sub-station to collect the measurement data periodically and to monitor the control commands that are executed on SCADA slaves. A global IDS runs in the control center and collects the measurement data from the sub-stations, estimating the overall grid state using the theory of differential evolution. The system also uses a reactive approach against the detection of attacks since every alarm triggered by the IDS notifies an intrusion

elimination system that in turn calls the local SDN controller to eliminate the asset from the network (Figure 3.7).

Fysarakis et al. 2017, present an industrial grade security framework equipped with both SDN and SCADA honeypots suited for being deployed to an operating wind park. The framework makes use of service function chaining (SFC), steering client traffic whose destination is the wind park between a sequence of virtual machines (a firewall, a general purpose IDS, a SCADA-based IDS, a deep packet inspector (DPI) and a traffic classifier) - see Figure 3.8 . The traffic classifier virtual machine built around a simple ACL, forwarded suspicious and/or not authorized network packets to a network of SCADA honeypots.

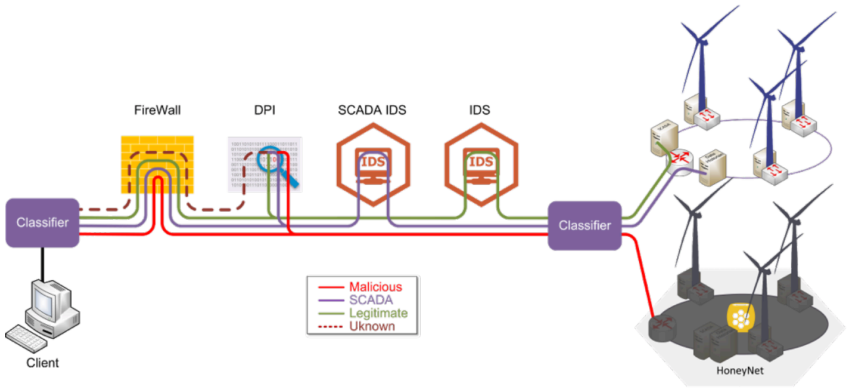


Figure 3.8: Reactive SFC framework proposed by (Fysarakis et al. 2017).

Using the OpenDaylight SDN controller and its SFC manager module, the authors conclude the system can greatly enhance the security of SCADA systems while keeping a low overhead in the network performance. In fact, albeit the network packets are chained through a set of virtual machines, sub-chaining pipelines can co-exist within a single SFC entry. Hence, only the packets detected as *unknown* by the firewall were forwarded to the deep packet inspection virtual machine and only SCADA-based protocol packets were forwarded to the SCADA IDS. It is also important to mention the proposed system used virtual machines created beforehand and did not rely on any kind of orchestration. Integration with OpenStack is planned as future work.

Despite all the solutions proposed to address SCADA security through the use of SDN, an important question does remain: *the centralized control plane by the means of the network controller can easily represent a single point of failure*. This is of significant importance in IACS since availability is often the most (and sometimes the only) important quality attribute industrial companies value. In fact, many of the solutions mentioned above rely on simple single node network controllers thus impractical to be used in production environments. Ghosh et al. 2016 studied how arbitrary failures in the control plane can affect the stability of the grid. Faults were injected in the control plane by both dropping network packets in the *controller-to-switch* secure channel or by causing excessive delays in the said link. The authors concluded the delays in the grid adjustment can reach seconds and thus cause significant degradation of the automatic gain control (a fundamental closed-loop control that regulates the electricity grid frequency) especially when the grid is in a transient state and experiencing large fluctuations in its system state.

Gyorgy 2016 argues SDN in IACS should use distributed controllers or, at least, redundant deployments in the network. This redundancy should not only be an availability requirement

in the point-of-view of the infrastructure but also as a way to protect the control plane against DoS attacks. The authors also point the necessity of adopting cryptographic suites (e.g. Transport Layer Security - TLS) in the path between the controller and the data plane.

Kurtz et al. 2017 also address this issue proposing a method comprising active redundancy regarding the network controller and a voting system to select the best decision regarding the programmability of the data plane. To avoid the unavailability of the controller the authors suggest to use multiple controllers in the same network. All of them receive a copy of status updates and incoming requests. In turn, their response is sent to a voting system that, upon receiving all the answers from the controllers, decides according to the majority and sends the most voted programming command to the data plane. In this way, a fail-over system is established for the controllers and it is possible to eliminate the propagation of orders issued by compromised controllers (as long as they do not reach a majority). This strategy is shown to meet the strict requirements of smart grids.

3.4 SDN-assisted security probe deployment

Network probes can be defined as a program or other device inserted at key locations of the network for the purpose of monitoring or collecting data about network activity. Examples of probes are intrusion and detection systems (IDS), network intrusion and prevention systems (IPS) or network honeypots. We have seen that, in traditional networks, network functions requiring packet monitoring often depend on dedicated appliances (Figure 3.9). When a new network policy is required (e.g. the deployment of a new IDS appliance) prior knowledge of the network configuration by the network operator is mandatory. The network operator has to have a high degree of expertise and extensive knowledge of all the device types since this normally means dealing with a multi-vendor network device environment (each of them having different configuration instructions). Even in the case an IDS is deployed through the configuration of a Switched Port Analyzer (SPAN) port, the operator has to deal with the management tools of a specific device. Moreover, as the network scales, problems with network contention at the link layer or the exhaustion of the monitoring device computational resources are likely to happen.

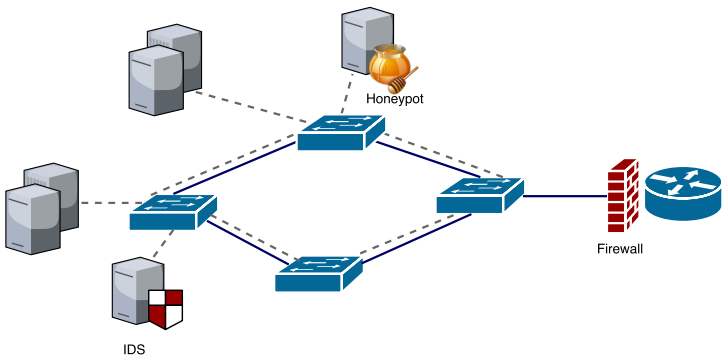


Figure 3.9: Traditional ICT network with the (manual) deployment of monitoring probes (IDS and Honeypot) – physical placement is key.

SDN, on the other hand, shows to be a promising approach for the deployment of virtual network monitoring assets (probes). The SDN controller provides effective network traffic

monitoring since it has direct or indirect control over the entire network topology. Also, the global awareness of the network state can enable the network controller to balance the traffic being monitored through different instances of an IDS thus avoiding the known contention problems of traditional deployments. Coupled with virtualization and orchestration technologies, the controller can also be used to instantiate and deploy new virtual machines for monitoring probes, greatly improving resource usage, similarly to the cloud computing paradigm.
The network controller, as seen in Section 2.2, has essentially two methods for flow instantiation: reactive and proactive flow instantiation. If the deployment of a network IDS is considered, it is quite easy to figure out how the network controller would program the network depending on the flow instantiation method.
If the controller uses a reactive flow instantiation approach the controller can simply forward the packet to the destination and to the running IDS host. In a proactive flow instantiation approach, logically sub-networks are created beforehand by the controller. So, when instantiating flow rules in the available network switches, the controller has to ensure the flow rule has two output ports: one leading to a path to the destination host and another leading to the IDS host.

Despite not directly applied to the context of IACS, there are several references in the literature which goal is to deploy probes (IDS and honeypots) recurring to SDN and the OpenFlow protocol. Those possibilities should not be discarded since their key concepts can easily be ported to other domains. Existing SDN-based network IDS solutions can be classified in terms of their implementation into two big groups (Lallo et al. 2017):

1. Those that implement the IDS as an SDN controller module.
2. Those that exploit SDN to easily forward traffic to dedicated IDSs.

If the mode of operation is taken into account, we can further expand those that belong to group (2) into two more groups:

- Those that use a reactive approach to receive the network packets in the controller and then offload a copy to a dedicated IDS.
- Those that simply modify the existing flow rules of the network switch fabric to also provide a copy of the traffic to a network IDS.

More recently, due to the advances in artificial intelligence, a few SDN IDS deployments that just exploit the statistical information of the controller are also starting to be proposed. Consequently, this subsection is aimed at exploring IDS and Honeypot solutions deployed through SDN and applied in other areas.

A literature review is provided next regarding existing IDS and Honeypot solutions as well as their deployments.

Yoon et al. 2015 point that the decoupling between data and control planes that SDN provides makes it inefficient for an IDS implementation since the network controller is only able to receive packet headers for the incoming packets at the L2 switch. According to the authors, a robust intrusion detection mechanism should have also access to the contents of the packet payload. The authors proposed to use the network controller to chain network traffic through a dedicated IDS as a possible alternative, although stating it would bring a performance overhead to the network.

Huang et al. 2015 used SDN as a mechanism to block attackers on the network, to prevent network scanning techniques and to prevent DDoS with the aid of an honeypot. An Openflow enabled switch (Pica8) was configured to have a monitoring port connected to a Snort virtual machine. Snort had a set of rules installed to detect botnets, network scanning operations or network flooding attacks and triggers the network controller each time an attack is detected (and an alert generated). The SDN controller installs a rule in the switch to drop the network connection of the attacker. In the case of a DoS attack, a webserver with weak security (acting as a honeypot) is added to the network as a means to detect attacks in Snort.

Z. Zhao et al. 2017 propose to use an IDS inserted in the path of the SDN switch fabric in order to detect host fingerprinting attempts. The system adopts the idea of the moving target defense to show hopping fingerprinters towards fingerprinting attackers.

Monshizadeh et al. 2017 also simply define a mirroring port in one of the network uplink switches but aim at a more scalable IDS. Hence, in order not to overflow the controller with network packets, the port is connected to a load balancer that distributes packets to multiple IDSs. The packet is only sent to the controller if it is considered malicious. An SDN application is then responsible to aggregate the malicious packets and to decide which flows to remove from the switch.

Jeong et al. 2014 propose a scalable intrusion detection system architecture on a SDN environment implemented using the Kernel Virtual Machine (KVM) infrastructure. In the proposed system all the network hosts are connected to OpenFlow-enabled devices and multiple IDSs (suricata) are pre-deployed on the network. Since the number of packets an IDS can process is limited the authors developed an optimization algorithm to calculate the sampling rate at which each switch should sample traffic while keeping the capacity of each IDS VM below its maximum value. Since the network controller stores statistical information regarding each of the flow rules, it was used to dynamically install rules resulting from the optimization model.

Shanmugam et al. 2014 also propose a scalable IDS for cloud environments (Figure 3.10). They argue that existing IDS/IPS solutions are inflexible and do not scale in terms of computing and networking resources.

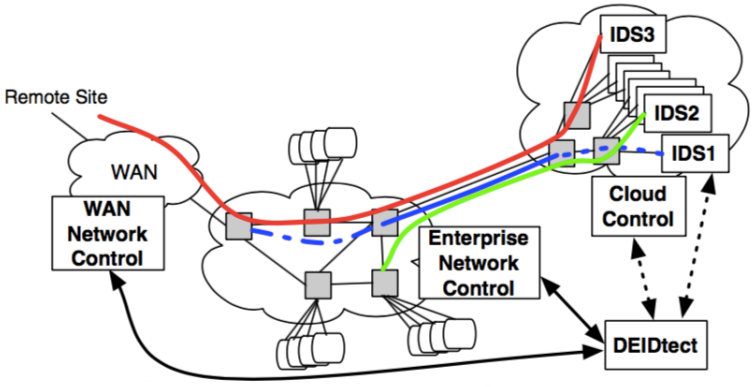


Figure 3.10: A scalable SDN based IDS (adapted from Shanmugam et al. 2014).

.

Furthermore, the authors also state there are use cases in which the traffic should be analysed by different IDS software packages. Hence, they propose to exploit SDN and OpenFlow along with the cloud computational model to modify the switch fabric pre-installed flow rules in order to provide multiple output ports for each rule. This is the typical example of an SDN-based IDS that applies a proactive flow instantiation approach.

Ajaeiya et al. 2017 propose to monitor SDN traffic with a lightweight IDS that periodically gathers statistical flow information from OpenFlow-enabled devices. *PACKET_IN* messages were collected in normal and "under-attack" conditions to extract flow headers and to train a classification model after feature aggregation. Feature aggregation such as packet and byte count, flow duration and byte count per duration ratio was performed in order to standardize a way of training classification models. The authors conclude Random Forests was the best classification algorithm, achieving a classification efficiency of 98% with minimal false positive rate. Additionally, the authors emphasize how this intrusion detection method has little performance overhead since it does not have any effect on the data plane and only takes advantage of the periodic statistical data asynchronous sent by the OpenFlow-enabled devices.

Abubakar et al. 2017 also follow a machine learning approach proposing a flow-based intrusion detection system to extend the already existing signature-based IDS (Snort) in a network star topology. Implemented on top of the network controller (and using its built-in REST interface), the system used a backpropragation algorithm to achieve a detection efficiency of 97%. Similarly to Ajaeiya et al. 2017, the authors did not change the network behaviour and exploited only the statistical data sent by the devices to the controller. The SDN controller device statistics are also used by Neu et al. 2017 as a way to detect insider attacks in encrypted communications between hosts in a SDN network. With no access to the packet payload, deviations to the usual network traffic patterns might be a sign of an undergoing attack. Thus, the authors state the controller itself can be seen as a lightweight IDS.

Trandafir et al. 2016 introduce an anomaly-based IDS and honeypot service developed with the closed-source Cisco implementation for SDN (onePK) that creates two different zones in the network topology: an IDS zone and an Honeypot zone. For the honeypot zone, the authors propose to use a controller as an "invisible" in-line traffic inspection appliance. All the network packets originating on the external network with the vulnerable honeypot virtual machines as the destination are sent from the network devices to the network controller. On the other hand, for the IDS implementation the network controller is not used in-line with the network traffic. All the internal network switches were configured to provide copies of the internal network traffic to an IDS virtual machine. The IDS VM made use of a machine learning algorithm trained with normal network patterns in order to generate signature-based rules for intrusion detection.

HoneyMix (Han et al. 2016) is one of the most important references on the use of SDN-based honeynets and serves as inspiration for similar proposals. The HoneyMix system was designed to leverage SDN to simultaneously establish multiple connections with a set of honeypots and to select the most desirable connection to inspire attackers to remain connected. Honeymix keeps a map of all available services in the network and implements multicast communication at the switch level for honeypot discovery. When a new connection is made targeting a specific protocol, a forwarding decision engine application is responsible for handling the forwarding logic. To accomplish that, all the network packets that arrive at the L2 switch fabric are forwarded to the controller. This application is aided by another

SDN application (behaviour learner) which computes the available load between the edge switches and all the available honeypots as a means to establish the best possible data stream. Since the selected connection is not always the same, another SDN application (connection selection engine) is responsible for re-writing the packet headers to successfully establish the connection. The system is shown in Figure 3.11.

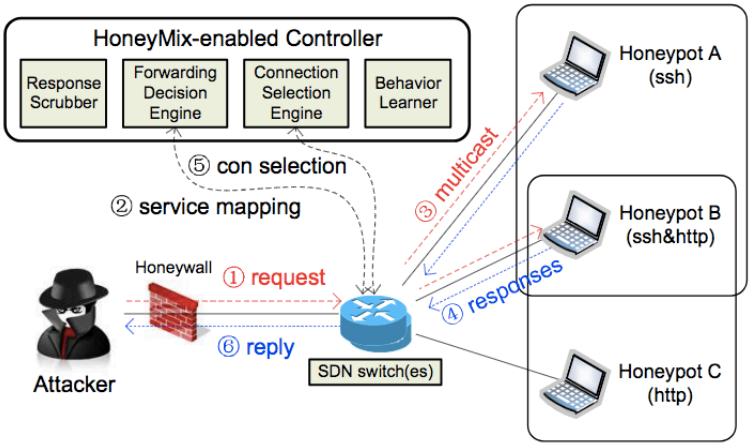


Figure 3.11: The HoneyMix framework for SDN honeypots (Han et al. 2016).

Manzano et al. 2016 show a prototype for an honeynet based on the SDN architecture, capable of detecting two types of network attacks: denial of service and network spoofing. Similarly to the reference above, the controller is also configured to receive and process every single network packet that arrives at L2. The controller runs a sequence of six modules against the received packet header to decide where to forward the packet.

3.5 On the use of NFV with container-based virtualization

Despite not available as a network driver option, OpenFlow support for Docker networking can still be enabled externally to the Docker API if we take advantage of the Linux Kernel namespaces (discussed in Section 2.3.1).

To the best of our knowledge, there is no published research related to container-based NFV in the context of IACS and critical infrastructures. However, a few references do exist which combine SDN and docker containers in other areas. These solutions should be evaluated and taken into consideration for the definition of architecture of this work.

It is of particular importance the work published by Cziva et al. 2016. The authors present GLANF, a framework for the deployment and management of virtual network functions in OpenFlow enabled networks using containers running common open-source utility binaries. An IDS (*Snort*), a packet filter (*Scapy*), a firewall (*iptables*), a traffic controller (*Tc*) and a load balancer (*Scapy*) were implemented in containers to which network flows were redirected by a network application installed in the *OpenDaylight* SDN controller. The authors conclude that container-based NFV improves function instantiation time up to 68% when compared with hypervisor-based alternatives. The authors explain why they used the Docker engine to encapsulate network functions in containers, pointing out advantages in line with the ones already mentioned above:

- It provided an easy way to encapsulate network functions in light-weigh containers with fast instantiation time, platform independent with high network throughput and low resource utilization.
- It provided a transparent way for network manipulation. Hosts in the network did not need to change their traffic patterns since traffic re-routing was handled by the network controller.
- It provided easy segregation mechanisms between data centre routing policies and the routing policies of the containers.
- NFs shared in public or private repositories alleviated redundant implementations and promoted collaborative development, contributing to the overall software quality of the product.

In order to achieve traffic redirection, Cziva et al. 2016 created a software daemon agent (GLANF agent) which included a REST API and was responsible for the instantiation of containers (using the Docker API) and for the attachment of running containers to the SDN network. Each instantiated container had two ethernet interfaces in the SDN network in order to perform service chaining. Network packets redirected from a network host enter the container through one of the interfaces, are processed by the software stack running inside the container and leave through the other interface (see Figure 3.12).

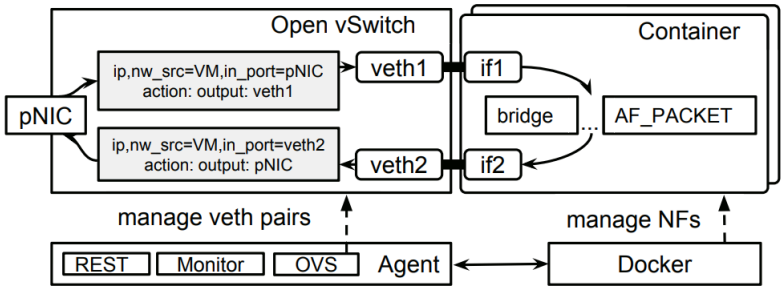


Figure 3.12: The GLANF agent proposed by Cziva et al. 2016 to attach docker containers to the SDN network.

The daemon REST API was exposed to a client application running on the SDN controller in order to issue commands for the creation and removal of containers on a given physical host. This agent also communicated with the Open vSwitch running on the physical host to create two virtual interfaces for each container and to attach them to a bridge controlled by the virtual switch.

No concrete information is provided regarding the way the agent attaches containers to the SDN network: only that the agent *communicates* with OpenvSwitch (OVS) directly. The OVS project provides a small bash script (see *OVS-Docker Github* 2017) which is able to create virtual interfaces and bind them to an existing Open vSwitch bridge.
Through a deep analysis of the utility source code it is possible to understand the *ovs-docker* workflow. Represented in Figure 3.13, when an container attachment request is issued by the user, the workflow of the *ovs-docker* utility follows the following logic:

1. The utility tries to find the PID of the running container (through the `docker inspect containerId`) command.

2. It then creates a network bridge in the OVS with the provided name if it does not exist already (`ovs-vsctl add-br bridgename`).
3. If the bridge exists, It creates a Linux virtual interface on the physical host (`ip link add type veth peer name`).
4. It attaches the virtual interface to the OVS bridge (`ovs-vsctl add-port bridge port`).
5. It moves the created virtual interface under the container PID namespace using the Linux kernel network namespaces (`ip link set port netns PID`).
6. The name of the interface is changed to a friendly name (e.g. *eth0*) and the interface is enabled (`ip netns exec PID ip link set dev PORTNAME name FRIENDLYNAME && ip netns exec PID ip link set FRIENDLYNAME up`).

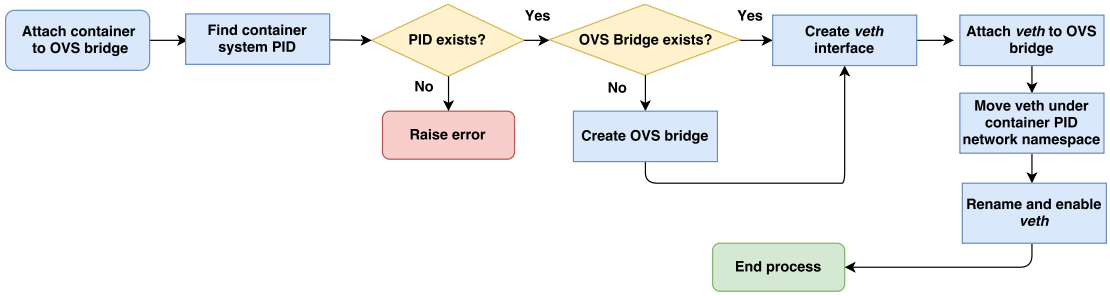


Figure 3.13: The *OVS-Docker* utility workflow.

Another important reference in this scope is the work proposed by Moradi et al. 2017, that created a distributed system (ConMon) to monitor container traffic in order to generate the network traffic matrix. Each physical host was composed by an instance of the Open vSwitch, a monitoring controller agent, a container management system (docker engine) and different types of containers: application containers and monitoring containers (active and passive) – c.f. Figure 3.14.

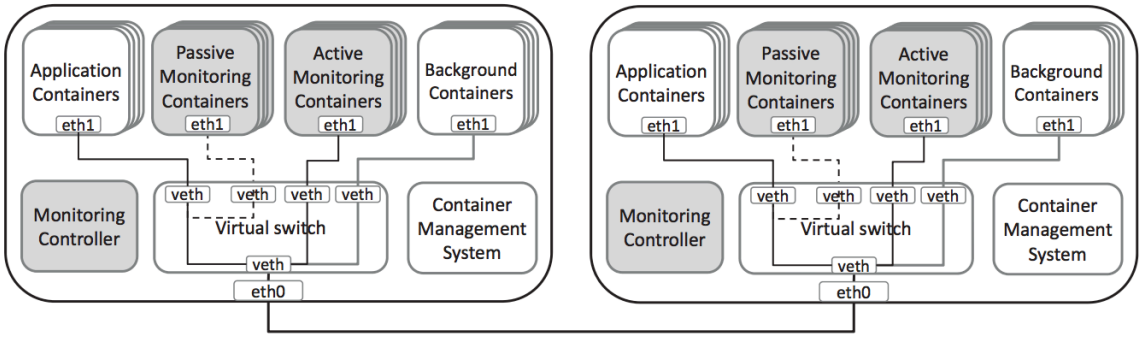


Figure 3.14: The ConMon multi-host scenario (proposed by Moradi et al. 2017).

Active monitoring containers communicate with each other in order to measure the overall network bandwidth. Passive monitoring containers receive copies of the network traffic

flowing between application containers through the installation of OpenFlow rules in the OpenvSwitch. The orchestration of monitoring containers is achieved by the monitoring controller agent which rely on the container management system to obtain container life-cycle events and to instantiate and remove monitoring containers. This monitoring controller agent is also responsible for attaching all the containers to the OpenvSwitch bridge of each host, for launching monitoring containers and to dynamically configure TAP ports on the switch itself.
Network data in the proposed system is stored in a distributed fashion using a distributed database shared between every physical host. Interestingly, this solution strips down the SDN stack: it neglects the need for a centralized network controller and relies the network orchestration to shared monitoring "controller" agents located in all physical nodes. Despite this fact, the idea of using a multi-host approach, the idea of relying on the docker engine container life-cycle events and the way the topology allows communication between physical nodes can easily be transferred to an SDN-controlled network and greatly inspire the proposed architecture of this thesis. Multi-host communication is achieved in ConMon by placing the physical network interfaces of the host (`eth0` in Figure 3.14) also on the OVS bridge. Since OpenvSwitch natively supports the OpenFlow protocol, the whole container networking can be controllable by manipulating the flow rules of the virtual switch.

3.6 State of the art overview and conclusions

Multiple relevant aspects are important to take into consideration after reviewing the available literature:

- Current SCADA architectures are segregated into multiple networks for security purposes. Within production lines, although a global view of the control process does exist, multiple sections can be considered somehow independent (each one comprising specific control equipment). Engineers, network administrators and process technicians cooperate together on a per-process level. So, in bringing SDN to the IACS domain this aspect should be contemplated. Multi-tenancy and stratification of the link layer must be considered a requirement.
- IACS is an area where innovation moves slowly. The drastic architectural change of redesigning the network via SDN should be attenuated by reusing security practices already well explored in this domain. Intrusion detection systems, honeypots and unidirectional gateways already exist in plant installations. Hence, if SDN comes to IACS it should be used only as a way to adapt and simplify existing security equipment instead of forcing a complete shift in existing security policies. New technologies come with the price of requiring education and training for process operators and pose a significant effect in the companies' financial budget.
- In process engineering, engineers rely in automation to keep a steady operation but still want to take the final decision in case an issue occurs in the production process. Operators are trained to perform specific actions depending on process indicators provided by the control loop. Plant equipment is designed with process troubleshooting in mind, often resulting from HAZOP (hazard and operability study) analysis. Similarly, if the infrastructure is under attack automatic reactions should be avoided as much as possible. Security mechanisms should be applied just as a way to detect and identify

possible threats and to provide feedback to network operators so a corrective action could be manually applied.

- Most of the suggested solutions that contemplate SDN in the IACS field are merely conceptual and often don't rely or explore the clustering nature of some SDN controllers. SDN is normally evaluated as a possible benefit to improve the resilience of the critical infrastructure but the effect of the control plane representing a single point of failure in the infrastructure is often neglected. Essential services value availability above any other attribute and, as a result, the big majority of the reviewed proposals would see difficulties in being accept by current industry stakeholders.
- It is common to see many frameworks create another architectural layer above the interfaces of the controller, e.g. using their REST APIs. This fact somehow violates the SDN architecture as critical logic requiring the controller services is shifted from the controller application layer. This results in a high number of HTTP calls between applications and the controller leading to low performance.
- Many SDN based IDS solutions are implemented as a controller module, thus forcing all the packet headers to be sent by network switches to the controller. This can easily be exploited in the form of DDoS attacks against the control plane. Furthermore, the advantages of SDN (through the definition of flow rules with multiple port outputs) are well-known when compared to IP multicast or switch SPAN ports. Recall that SCADA network flows are somehow static and the connection matrix is (most of the time) known beforehand. Hence, network intrusion and detection systems in IACS can take advantage of the OpenFlow protocol to build more intelligent and dynamic solutions while lowering effects on the network performance at the same time. Proactive flow instantiation seems to be the correct approach to build an IACS network intrusion detection system.
- Container based virtualization shows to be promising in reducing the time required for creating new network assets, in optimizing computational resource usage and in simplifying the workflow required to deploy (and version) new software. There are almost no references in the literature that take advantage of containerization and SDN. Specially, as far as we know, the usage of containers in IACS was never evaluated nor implemented.
- Honeypot and IDS solutions present in the literature do not contemplate orchestration nor automatic deployment. Scalability is often pointed as an advantage for SDN IDSs (in the form of use cases) but no real world deployments or solutions exist.

The outcome of this thesis can then be considered important as it proposes to address the above mentioned issues. We propose to develop a set of SDN applications in a distributed network controller (favouring availability) and take other design decisions to improve the performance of the system. Using Docker containers for network function virtualization and a proactive rule instantiation approach lowers the impact on the control plane and reduces resource usage in the virtualization infrastructure. Using virtualized versions of IACS probes deployed through a common web-management interface and storing all the probe images in a common registry location bring unprecedented flexibility for IACS operators. Taking advantage of the topology graph served by the SDN controller helps the deployment process and reduces the barrier created when shifting from physical deployments to virtualized instances consolidated in the virtual infrastructure.

Chapter 4

Requirements

This chapter details the elicited requirements (and the requirements elicitation process) for this thesis. Since the work of this thesis emerges in the context of an Horizon H2020 project (the ATENA project), subsection 4.1 provides an overview of the platform proposed by the University of Coimbra (in which the work of this thesis is applied). Section 4.2 presents the elicitation process detailed for each of the elicited requirement types. Subsection 4.2.4 provides the product perspective. Within the section, the system actors, the functional blocks of the so. Afterwards, the elicited requirements are summarized:

- Section 4.3 details the elicited functional requirements.
- Section 4.3 presents the non-functional requirements.
- Section 4.5 is due to the design constraints of the system.

Finally, Section 4.6 provides a chapter wrap-up providing a synthesis of the requirements elicitation process. Functional requirements were collected in the form of use-cases. The use-case diagrams can be found in Annex A of volume II. Use-case descriptions for each use-case can be found in Annex B. Note that this section is a result of a joint work and was published in the project internal documentation (AtenaConsortium4.1 2017).

4.1 The Intrusion and Anomaly Detection System - IADS

This thesis emerges in the context of the ATENA project, an European research effort which aims at addressing the interdependencies between critical infrastructures and at proposing new tools to access and mitigate the effects of ICT components on the critical infrastructure. Within ATENA, the University of Coimbra (UC) proposes to develop an Intrusion Anomaly and Detection System (IADS) responsible for the detection of unwanted and/or unauthorized actions such as those resulting from cyber attacks. This component is strongly connected with other parts of the ATENA architecture either for receiving events (from probes) or to generate security alerts which can then be further processed by the ATENA architecture upper layers. The IADS module is responsible for monitoring the underlying critical infrastructure environment by recurring to distributed probes. Probes, in the IADS context, can be perceived as regular ICT network probes (e.g honeypots, honeynets, network intrusion and detection systems) or specific physical domain probes (e.g. shadow RTUs responsible for monitoring the process control I/O channel and reporting deviations to the expected PLC response (Cruz et al. 2015)). Probes are physically dispersed in the critical infrastructure and generate events for suspicious activity. Those events are processed by the IADS platform

before being sent to the upper layers of the ATENA architecture (which is then responsible for classifying all the alerts generated by either the IADS or other ATENA components).

To better understand the context in which the work of this thesis is being applied, the IADS reference architecture is presented in Figure 4.1.

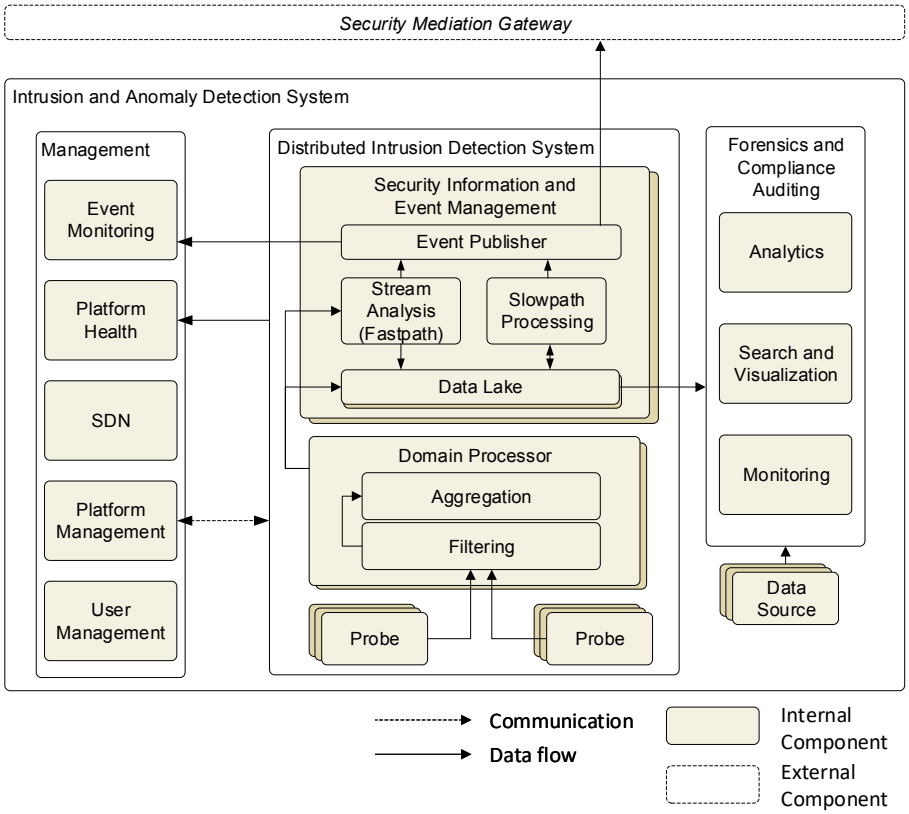


Figure 4.1: The ATENA *Intrusion Anomaly Detection System* (IADS) reference architecture

.

As seen in Figure 4.1, the IADS reference architecture is meant to be highly distributed, indulging the infrastructure availability above any other type of requirements. The IADS architecture includes several components, such as (Graveto 2017):

- Probes - The "eyes" of the critical infrastructure, ranging from conventional network and host security components (e.g. network IDS) to IACS field-specific probes. Probes generate events to a domain processor using a established data model (AtenaConsortium4.3 2017) and the Apache Avro encoding.
- Domain processor - Clustering brokers per probe scope, backed by a Message Queuing system. The domain processor is used to receive all the events generated by distributed probes and to apply simple and quick processing to events (e.g. aggregation, filtering, time-windows).
- SIEM - The "brain" of the platform, the *Security Information Event Management component* is used to support streaming and batch processing of the received events. Several machine learning algorithms and specially crafted applications run on the SIEM

to decide if the events generated by the security probes are in fact a result of an ongoing cyber attack.

- Data Lake - A distributed database used to store all the data transiting the system. It is useful as the data source for SIEM processing and also to allow forensics on the infrastructure data.
- Forensics and Compliance Auditing (FCA) - The component responsible for enabling post-mortem data analysis on the incidents of the critical infrastructure or ongoing compliance validation of organizational security policies.

Each of the above modules is built on a distributed fashion inspired by *Big-Data* principles. The architecture is designed to accommodate and scale in/out according to the specific needs of the infrastructure being monitored (and protected) by the IADS since the replication factor of each of the IADS components highly depends on the specificity of each critical infrastructure (i.e. number of events, sources, etc). In the ATENA IADS architecture, all the events sent to the domain processor are encoded using the Apache Avro binary format (for increased message transfer performance). The domain processor component is supported by clusters of Apache Kafka (backed by the Kafka Streams API for data processing while receiving events). The data lake component is built on a cluster of Cassandra NO-SQL database nodes. The IADS SIEM is built over the Apache Spark framework aided by its internal MLlib (for machine learning).

4.1.1 The role of SDN/NFV in the IADS platform

Despite the completeness of the architecture presented in Figure 4.1, there are a few important missing pieces, namely the orchestration and the management of distributed probes. Furthermore, all those actions are normally performed on a common management dashboard. In the IADS context, probes can be either physical (e.g. physical network TAPs in-line with the IACS network equipment) or virtualized versions of such probes (virtual machines or containers) running on common-of-the shelf hypervisors or physical hosts. While probe management falls a bit out of the scope of SDN (and is more a role of protocols such as MQTT, CoAP or management toolsets as Apache Leshan), the orchestration, provisioning and network programmability of such virtual probes are key aspects which can only be enabled by coupling SDN with NFV. If we take a virtualized version of a network IDS as an example, the simple deployment of a IDS container (or virtual machine) does not make every network packet to reach the launched container. A SDN application is required to program the switch fabric so that copies of the network packets can reach the instantiated container. The same happens with any virtual version of an honeypot. The automatic deployment of an honeypot can only be viable if along with the container/VM deployment specific rules are installed to forward network traffic to the launched asset. The network controller provides easy access to the global network topology making it easy to program the network traffic matrix (authorized host-pairs and respective protocols), to create sub-logical networks based on host-pair communication and to define what happens on a specific network link (block or allow traffic). As a result, SDN is perfect to combine probe instantiation (e.g. the creation of a container) with network programming to enable the probe operation (network orchestration), since the global awareness makes the location of the probe negligible. Furthermore, the global network topology provides SDN with fine grained control over which devices to monitor and the locations for specific services to be deployed (e.g. choose the

edge link that operates as a data diode). Network controllers also receive network statistics from OpenFlow enabled switches (along with other controller specific events) allowing custom applications to also operate as probes for the IADS platform. Network controllers also have REST API's and often allow extending those API's. As a result, this information can be exposed to a management dashboard where network operators will be able to deploy services.
The main goals of SDN in the ATENA project (and inherently in this thesis) are identified below. These goals are useful to segregate the SDN subsystem of the IADS platform by functional blocks.

1. **Logical sub-networks** - Leverage SDN to create a multi-tenancy network. Logical network should be created in the overall network topology and associated with specific user (tenant) accounts.
2. **Virtual IDS** - To develop an SDN application that enables the existence of virtual IDS services. An IDS in the context of this thesis is not related to the detection capabilities of the probe itself but instead with the network programmability that makes achieving the said probe operation possible. The application should be able to instantiate a new container based on a IDS image template (e.g. Snort), provide copies of the traffic to the container and scale the number of containers according to the network load or container resources constraints (e.g. CPU utilization).
3. **Virtual Honeypot** - To develop an honeypot SDN application that is able to instantiate a container (with a specific image) and specify a range of IP addresses for the honeypot container operation. The application must program the network so that any connection attempts to the said IP addresses are redirected to the honeypot container.
4. **Data Diode** - To develop a data diode application based on SDN. The application should allow the selection of a given edge link in the network topology and block any traffic on one of its directions. In fact, the Evaluation Assurance Level 7 criteria (EAL7) requires the use of data diodes (uni-directional gateways) between networks as a security mechanism (FortFox 2010).
5. **Network Event factory probe** - To develop a probe (SDN application) that forwards any SDN event (controller, device, host, link, statistics) to a domain processor. The application must encode the event in Avro using the ATENA datamodel.
6. **Management and visualization web-interface** - To concurrently develop a web-interface where network tenants (and operators) can deploy and monitor the mentioned network services. Ideally, it should make use of the network topology graph to help visualizing service (and container) deployment.

4.2 Requirements Elicitation

This section explains the methodology followed for the elicitation of requirements in the context of the IADS system and their sub-components. Subsection 4.2.1 contains the elicited requirement categories and their definition. In subsection 4.2.2 the conventions followed to systematize and identify all the requirements in the current document are explained. The actual methodology followed for the elicitation of the requirement types is discussed in Subsection 4.2.3.

4.2.1 Requirement types

IEEE Std 1233 1998, defines a requirement as:

1. a condition or capability needed by a user to solve a problem or achieve an objective.
2. a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. a documented representation of a condition or capability as in (1) or (2).

In the specific context of a software system, requirements are the description of services that a software piece must provide and the constraints under which it must operate. These can be divided in:

- *Functional requirements* - they relate to system features, i.e., the statement of services the system should provide, how the system should react to a particular set of inputs and how the system should behave in particular situations. They are often further classified on user and system requirements. User requirements come from a system actor or any other type of stakeholder and express a property of the domain that the introduction of the new system will bring. On the other hand, system requirements express a desirable system property that when implemented will lead to at least one user requirement.
- *Non-Functional requirements* - Also known as quality attributes they define the characteristics on the services or functions offered by the system. These might be related to timing constraints, security, to the development process or standards. They define design and implementation constraints and usually apply to the system itself.
- *Design Constraints* - Usually hard to define and tending to be "nebulous", they refer to requirements the system should follow in order to ensure the system complies with the specified functional requirements or system goals.

Figure 4.4 illustrates how the elicitation of the different types of software requirements contribute to the overall system design.

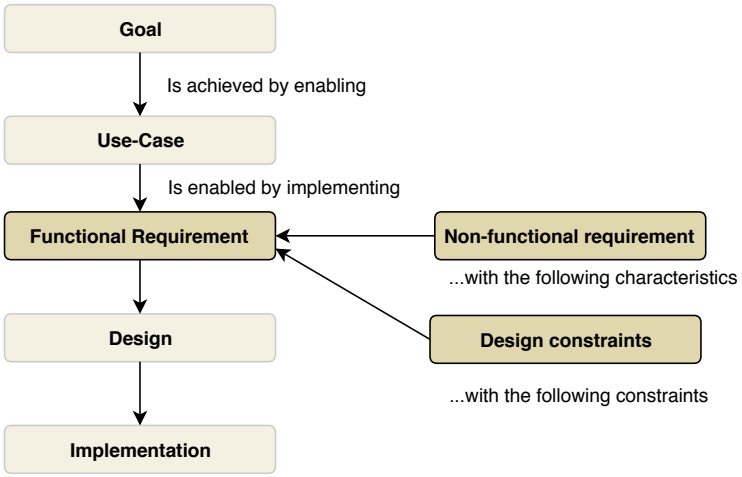


Figure 4.2: Software requirements contribution to the overall system design (adapted from Williams 2006).

Apart from the above classification of requirements, there are few common characteristics that make a "good" requirement:

- Minimal – means that only the necessary requirements are stated so that the design space is not restricted prematurely.
- Complete – means that all the requirements of the stakeholders are captured.
- Focused – means that the impact of the requirement on the solution is clear. This definition supports unambiguity in the sense of the IEEE Std. 830 (IEEE 1994).
- Measurable – especially important in the context of non-functional requirements, means that a metric is given on how to verify that the system satisfies the requirement. This supports verifiability and unambiguity in the sense of the IEEE Std. 830 (IEEE 1994).
- Traceable – rationales are given that describe why the non-functional requirement (NFR) is necessary and how it is refined into sub-characteristics. This also supports modifiability in the sense of the IEEE Std. 830 (IEEE 1994).

4.2.2 Requirement conventions

Table 4.1 shows the structure used in this section to present the elicited requirements. This convention was followed in order to ensure consistency across the presentation of all requirements independently of their type.

Table 4.1: Requirement presentation.

ID	Requirement	Priority
#id	#use-case name and description	#use-case priority

The table contains a brief description of each requirement, its identification (ID) and its respective priority. The rationale used for the definition of each parameter is presented below.

ID

The requirement identifier follows a logical convention, e.g.:

`PackageName_RequirementType_RequirementNumber`

where the package name identifies the context of the requirement and is mapped with a specific system package, namely:

- Users Management (**UM**),
- Network Management (**NM**),
- Network Event Factory (**NEF**),
- Container Management (**CM**),

- Virtual Network Intrusion and detection system (**VN**),
- Virtual Honeypot (**VH**),
- Network Statistics (**NS**),
- Container Statistics (**CS**).

The requirement type clearly identifies if the requirement is a functional requirement (FR), a non- functional requirement (NFR) or a design constraint (DC). The requirement number unequivocally identifies the requirement inside each package. As an example, the requirement UM_FR4 means the functional requirement number four of the package Users Management.

There are, however, some cases in which the requirement is transversal to multiple system packages and is better applied to the whole system itself. In those cases, IADS is used instead of the package name (e.g. IADS_DC1 – The IADS system design constraint number one).

Priority

Since the project has a fixed deadline, the MoSCoW prioritization technique was used in order to reach a common agreement on the importance of each requirement. The MoSCoW method defines four well established prioritization categories (adapted from Taylor and Mead 2016):

- Must have (**MH**) – Requirements labelled as *Must* have are critical to the current delivery. If even one Must have requirement is not included, the project delivery should be considered a failure.
- Should have (**SH**) – Requirements labelled as *Should* have are important but not critical for delivery. While *Should Have* requirements can be as important as *Must have* there might be another way of satisfying the requirement.
- Could have (**CH**) - Requirements labelled as *Could have* are desirable but not necessary as they often represent possible user experience improvements. These are included if time and resources permit.
- Won't have (**WH**) - Requirements labelled as *Won't have* were considered by the project stakeholders as the least-critical and, as a result, are not part of the delivered implementation.

4.2.3 Methodology

The requirements elicitation represents an active effort to extract information from stakeholders and subject matter experts. It does not represent a step or a task within the development process but defines a set of techniques to be applied during the requirements phase. In some software development life-cycles, the requirements phase might follow the whole life-cycle. Even though the techniques used to elicit requirements are different depending on the requirement type, a standard pro-active deliberated search approach was followed throughout all the elicitation process as illustrated in Figure 4.3 (Taima 2014).

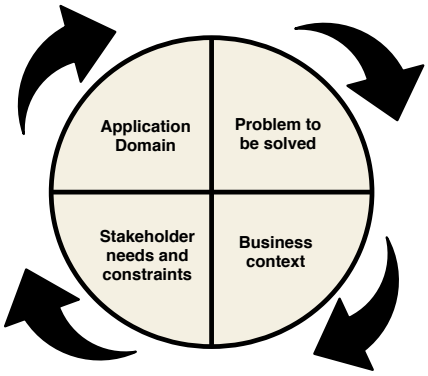


Figure 4.3: Requirements elicitation process (adapted from Taima 2014).

The iterative process started with the effort to understand the application domain, i.e., the knowledge of the general area where the system is to be applied (Industrial Automation and Control Systems). After understanding the application domain, it was important to correctly define the problem to solve (problem understanding). This step was followed by the correct understanding of the business area of the problem. By doing this, it was possible to determine stakeholders and their interactions (business understanding) and to map them to system actors. The last step of the elicitation process was to actually discuss and gather the requirements that apply to each one of the system actors. That implied the correct understanding of the specific needs of the people who require the existence of features on the system to be designed and developed.

Throughout the accomplishment of the all the process elicitation steps, several auxiliary techniques were used to elicit and gather the requirements:

- **Brainstorming** – Generating creative ideas and reason about the overall system and the respective solutions through intensive and free-wheeling group discussions.
- **Process Modelling** – Understanding the system scopes, actors and their relationships in a way the system can be reasonably explained to stakeholders not familiar with the system (use-cases mentioned in the next subsection "Functional Requirements" are an example).
- **Prototyping**
  - Visually representing parts of the user interface in order to understand the organization of the information system scope, possible metrics to be monitored and filling uncovering gaps.
  - Building early stage versions of the software in order to better understand how some protocols work or the requirements of other software components the system depends on.

Functional requirements

For the elicitation of the cyber-physical IADS functional requirements the Use-Case definition approach was followed. Use-cases are a formal methodology to document functional requirements that, for each of them, provides a list of actions or event steps. These steps

typically define the interactions between a role (or the actor as it is known from UML) and a system in order to achieve a final goal. The use-cases are very useful and provide farther more information than other less-formal functional requirement elicitation methods such as user stories (that just describe a brief story and the final goal). Thus, in the software development lifecycle, use-cases provide the developers a greater level of system detail prior to the development and implementation phases. A use-case also helps to identify the system's exceptions to the expected success scenario and the interactions between the system roles and the various system functional scopes. Furthermore, use-cases provide the needed software artefacts for the stakeholders to reason about the system in the context of the definition of the software architecture. In other phases of the software development process lifecycle such as software testing, use-cases are also a valuable asset.

During the functional requirements elicitation phase the use-case definition approach has made clear some other important advantages, such as (J. Goss 2007):

- Helping to improve the communication between team members.
- Encouraging the reachability of a common agreement about system requirements.
- Revealing process alternatives, process exceptions and undefined terms.
- Exposing what belongs outside a project scope.
- Transforming manual processes into automated processes.
- Recognizing patterns and contexts in functional requirements.
- Helping to prioritize work.
- Helping to discover gaps between the requirements and the expected software to be delivered.

Each use-case description followed a common template as exemplified in Table 4.2. This table contains the definition of all of its fields.

Table 4.2: Template for use-case description.

Use-Case	**ID**
Primary Actor	The role name for the primary actor that has the main responsibility for this use-case
Secondary Actors	The role name of another actor(s) that have permissions to use it. (It is not a mandatory field for some use-cases)
Scope	The name of the design scope where this use-case is integrated
Level	The use-case level (Summary-Goal, User-Goal, Sub-Functions)
Stakeholders and Interests	List of all interested stakeholders and key interests in the realization of the use-case
Pre-Conditions	All pre-conditions that must to be have successful executed before its use-case is triggered and that we expect that they are the state of the world

Last Review	The last review date of this use-case
Minimum Guarantees	All minimum guarantees that must be ensured in case the main success scenario fails
Success Guarantees	All success guarantees that must be ensured if the main success scenario ends successfully
Trigger	The main action that starts the use-case
Process - Main Success Scenario	All steps that describe the process of the main success scenario of this use-case (upon successful completion)
Exceptions	All exception descriptions if an exception/error occurs in a step of the main success scenario

Table 4.2 has all the use-case fields in the left column and the respective description in the right column. All these fields are useful to guarantee that a clear understanding of the needed process steps to achieve the use-case goal, the interested stakeholders, the pre-conditions that must be met before the use-case starts and the exceptions to the success scenario. Although a few fields might need additional explanation. The level field can be one of three types: summary-goal, user-goal and sub-functions. A summary-goal involve multi user-goal levels, i.e., cannot be completed in one sitting and may require multiple people, organizations, and systems interacting to achieve the goal. A user-goal is in the level of a greatest interest representing the goal of the primary actor trying to get the work done. A sub-function (also represented in the diagrams by a clam symbol) is a use-case representing a low-level system requirement needed to carry out a user-goal task. Use-cases are summarized in use-case diagrams respecting the UML modelling language (see Annex A (Volume II). Due to their low-level, sub-functions usually are not detailed in the form of use-cases; they are only present on the use-case diagrams in order to help understanding the system and the relationship between use-cases (Cockburn 2000; K. M. Anderson 2005). It is also important to mention that some of the UML relationships used in the definition of the use-case diagrams shown in the figure below and used in the diagrams of Annex A (Volume II).

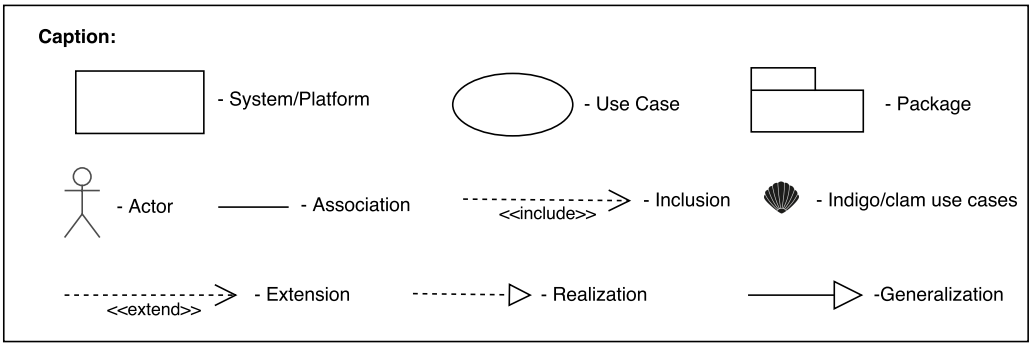


Figure 4.4: Use-case relationships. Caption used for the use-case UML diagrams of Annex A (Vol. II)

.

- Association – Used to associate a specific use-case with a given system actor.
- Inclusion – Base use-case is incomplete and the included use-case is required, not optional.
- Generalization – Base use-case could be abstract (incomplete) or concrete (complete). The specialized use-case is required, not optional if the base use-case is abstract. Generalization may also be applied for actor relationships defining their inheritance.
- Extension – Base use-case is complete by itself and can be defined independently. The extending use-case is optional/supplementary.
- Realization – Special abstraction relationship between two use-cases. The base use-case is abstract and the realization use-case is an implementation of the first, i.e., a concrete use-case.

Non-functional requirements

The quality attributes of a system define an important class of non-functional requirements. They concern software system attributes such as functional suitability, performance, availability, security and are important for achieving stakeholder goals (Fotrousi et al. 2014). ISO/IEC 2010 defines a quality model in order to identify the degree to which the system satisfies the stated and implied needs of its various stakeholders. This model relates to both static and dynamic properties of computer and software systems. It categorizes non-functional requirements in eight attributes (which in turn are also divided in sub-characteristics):

- Functional suitability
- Performance efficiency
- Compatibility
- Usability
- Reliability
- Security
- Maintainability
- Portability

Meeting the right level of quality is important to balance benefits and cost (Regnell et al. 2008). The quality definition of a software system needs to be good enough to make the software useful but not so excessive that makes its financial and resource costs impracticable. In addition, some quality attributes often reveal to be incompatible. A good example is the CAP theorem (Consistency Availability Partition tolerance), also named Brewer's theorem, usually applied to distributed data stores (Brewer 2000).

The most known method of eliciting non-functional requirement is based on the definition of scenarios. These scenarios are small stories that provide a global framework for systematizing non-functional requirements since they include:

- Stimulus (a fault happening to the system).

- The source of the stimulus (internal or external).
- Environment (the state of the system when the failure occurs).
- Response (the reaction of the system to the failure).
- Response measure (the response of the system to the failure as a specific measure – percentage, downtime, mean time between failures, etc.).

A graphical representation of the scenario approach to elicit non-functional requirements is presented in Figure 4.5.

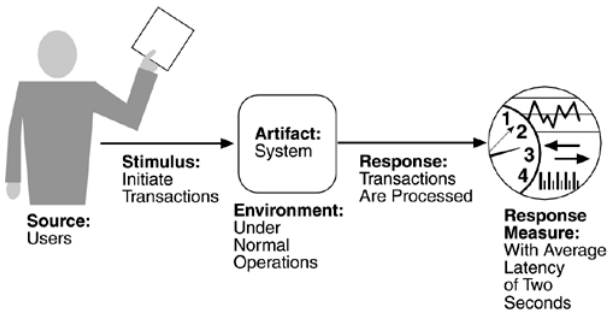


Figure 4.5: Scenario based approach for the elicitation of non-functional requirement (reproduced from eTutorials.org 2008).

Since the scenarios elicitation approach requires concrete metrics to validate each quality attribute (and these metrics are not always clearly identifiable) an alternative approach similar to the one introduced in Fotrousi et al. 2014 was followed. The elicitation method for non-functional requirements was based on inquiries between stakeholders with the main focus of understanding the relationships between quality attributes and their impact on the system. The method was composed of four iterative steps (which could have also been composed of several rounds) and had its main strength on the construction of small prototypes :

1. **Preparation**: Construction of prototypes and other materials needed to allow the stakeholders to experience the quality attributes under investigation.
2. **Measurement**: Workshops were performed with some stakeholders with the aim of collecting quality measurements and user feedback. They were especially important to reason about the system quality.
3. **Analysis**: Correlation between the collected feedback and the several opinions on the impact of each quality attribute on the system.
4. **Decision-Making**: Decision regarding the system quality attributes and whether to evolve a given attribute to specific requirements for the requirements document.

The result of the process above led to the definition of four main quality attributes for the IADS system:

- **Security**: Measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users. Security can be characterized as a system ability to provide non-repudiation, confidentiality, integrity, assurance, availability and auditing.

- **Performance**: Indication of the responsiveness for the system to execute actions within a given time interval.
- **Availability**: Availability quality attribute is concerned with the system failure and its associated consequences. A system failure occurs when the system no longer delivers consistent service with its own specification.
- **Interoperability**: the ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties. An interoperable system makes it easier to exchange and reuse information internally as well as externally (Microsoft 2017).

Other non-functional requirements that are not directly mapped to any of the quality attributes presented above were also identified. Those requirements were grouped in a category named "Operational and Environmental requirements" (see Section 4.4.4) and relate to either the deployment of the system within the IACS infrastructure or the operational environment of the system itself. This category of requirements limits the effect that the external environment has on the system and/or the effect the system is to have on the external enveloping environment (Tinsley et al. 2006). Each of the quality attributes were evolved to specific formal requirements. Those requirements and the respective mapping to the quality attribute are presented in Section 4.4.

4.2.4 Product Perspective

This section is aimed at giving a brief overview over the system context, its actors and subsystems.

4.2.5 System actors

An actor represents a division of system behaviour defined by the role played by an external entity that interacts with the system through the exchange of signals and/or data. The role is often used informally as a particular user group that require specific functionalities or properties from the system. Any external entity interacting with the system is said to play the role of a defined actor. In order to meet the goals of the IADS system, several actors were identified. The actors are differentiated in terms of frequency of use (i.e. the work done by the system to answer functions for the specified user group), subset of available functions and privilege levels. The user privilege level builds up on a hierarchical fashion meaning each actor may inherit privileges/ability to perform actions in the system from another user. Table 4.3 presents the system actors, their need for the system design (role/rationale) and their level of privilege. The level of privilege is directly related with the possibility of the actor to execute administration tasks.

Table 4.3: System actors.

Actor	Rationale	Privilege Level
System Admin	Actor has full administration and control over all the cyber-physical IADS platform	HIGH

Security Admin	Actor has full administration access only to the security components of the cyber-physical IADS (e.g. Probe management)	HIGH
Security Monitor	Actor has monitoring privileges over all the cyber-physical IADS system including the virtual infrastructure. However, he cannot perform any administration action (i.e. any action that changes the system state such has the deployment of new probes)	MEDIUM
Network Admin	Actor responsible for managing or administrate the virtual infrastructure (SDN). This includes create and manage sub-networks, assign them to network tenants and perform all the actions a network tenant has access to.	HIGH
Network Tenant	Actor has a logical sub-network in the virtual infrastructure and is able to deploy virtualization services (virtual NIDS (vNIDS), virtual Honeypots (vHoneypots)) or apply software defined networking functionalities (e.g. set a network link as a data diode) to his own sub-network.	LOW

4.2.6 System functional scopes

Functional scopes, used in the context of use-cases, refer to the services the system offers. They are elicited at the same time use-cases are being written as it is not always an easy task to pointedly define them or to draw a boundary between which functionalities belong to each scope. In the case of the Cyber-Physical IADS, if seen as the overall system to be developed, it is quite easy to infer it is internally divided in several subsystems. Table 4.4 shows the different subsystems that compose the IADS system.

Table 4.4: System functional scopes.

Functional scopes	System	Subsystem
IADS	✓	
SDN		✓
Probes		✓
Event Streaming Platform		✓
Domain Processor		✓
SIEM		✓

In the scope of this thesis, all elicited requirements are under the SDN scope (Table 4.4) of the IADS platform. The description of all scopes that compose the IADS system is detailed below:

- **SDN** – Scope that represents the virtual-infrastructure controlled with a software defined networking approach.
- **Probes** – Virtual or physical elements that analyse available information and push relevant events to the domain processor component.
- **Event Streaming Platform** – Scope for the component responsible for consuming events and to distribute them internally within the IADS system.
- **Domain Processor** – Scope related with the component responsible for consuming events from the event streaming platform, executing data pre-processing logic and push the processed events to the event streaming platform.
- **SIEM** – Scope that receives events from the domain processor, persists the events in a data lake and adopts the slow-path and fast-path big data processing approaches on the received events.

Also in the context of use-cases and when the scope is mentioned it is common practice to clarify if the respective use-case is viewed as a black-box or as a white-box use-case in relation to the scope. In other words, a use-case that defines a process in which the only interest is to describe the actor inputs and the correspondent system output (i.e. the actor interactions with the system) is viewed as a black-box use-case. If, otherwise, the use-case description specifies the system behaviour in a way that defines how the system implements the interactions internally, it is viewed as a white-box use-case. It is a good principle to avoid the definition of too many white-box use-cases to avoid restricting the system design phase and to block completely the modifiability of a given requirement (refer to Section 4.2.1).

4.2.7 System packages and context diagram

When a use-case model is structured there are advantages on thinking and organizing it into smaller units, since it makes easier to show relationships between the model's main domain and the way the system is decomposed. To accomplish the decomposition, use-case packages are often used. Each package is a portion of the global use-case model and is composed by a semi-independent collection of closely related use-cases. It is also easy to understand that it is possible to have multiple levels of use-case packages, depending on the complexity of the software system (IBM Corp 2006).

The IADS system is divided in many packages for clear understanding and organization. Starting by Figure 4.6, it is possible to see the IADS system is composed by two "big" packages called *Management* and *Monitoring*. These two big packages divide the IADS system in terms of management functional tasks and monitoring tasks. Each of these are also composed by more sub-packages. The system context diagram presented on Figure 4.6 shows the connections between the system packages and the respective actors. Colored packages aggregate use-cases developed in the context of this thesis.

As shown in Figure 4.6 the IADS platform has a management and monitoring domains. The management domain can be further decomposed into the following sub-packages:

- **Users_Management** - use-case package responsible for the management of platform users. Since the SDN subsystem of the IADS platform requires the existence of different roles (Network Admins, Network Tenants and Security Monitors) users have to be also registered in the SDN controller so functionalities in the SDN domain can be segregated into a per user type level.
- **Platform_Management** - Out of the bounds of this thesis, this use-case package is related to management of probe components, domain processor, event stream and SIEM.
- **Virtual_Infrastructure_Management** – this use-case package is the core of this thesis and relates to management of containers, logical sub-networks and security services deployed on the SDN network.

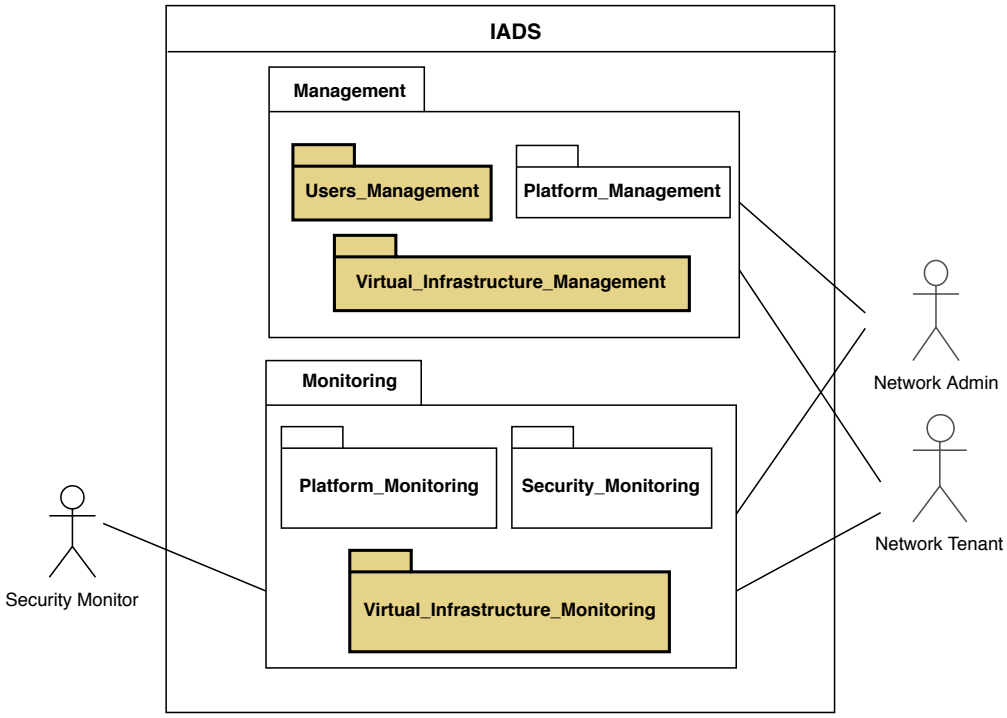


Figure 4.6: IADS use-case context diagram.

Figure 4.7 details how the Virtual_Infrastructure_Management use-case package is decomposed into several other use-case packages that group use-cases belonging to the same domain. This use-case package is split into three other sub-packages:

- **Network** - Use-case package that groups all networking related requirements related to the section of the IADS platform controlled via SDN. The package can also be further divided in two more packages:
  - Network Management - The management of the SDN network. The package includes all the use-cases which are related to the creation of sub-networks, association of sub-networks to network tenants, accessing the network topology and/or listing of network assets (hosts, links, devices).

  - Network Event Factory - Groups all the use-cases related with the SDN probe which goal is to send SDN related controller events to the upper layers of the IADS platform.

- **Containers** - Sub use-case package which groups all the requirements which relate ti the management of container applications and its supporting infrastructure.
- **Services** - Management package for the different virtualized network functions operating in the SDN network. This includes:
  - **vNIDS** - Scalable network IDS service requirements.
  - **vHoneypot** - Virtual honeypot deployment service.
  - **Data Diode** - Virtual data diode service requirements.

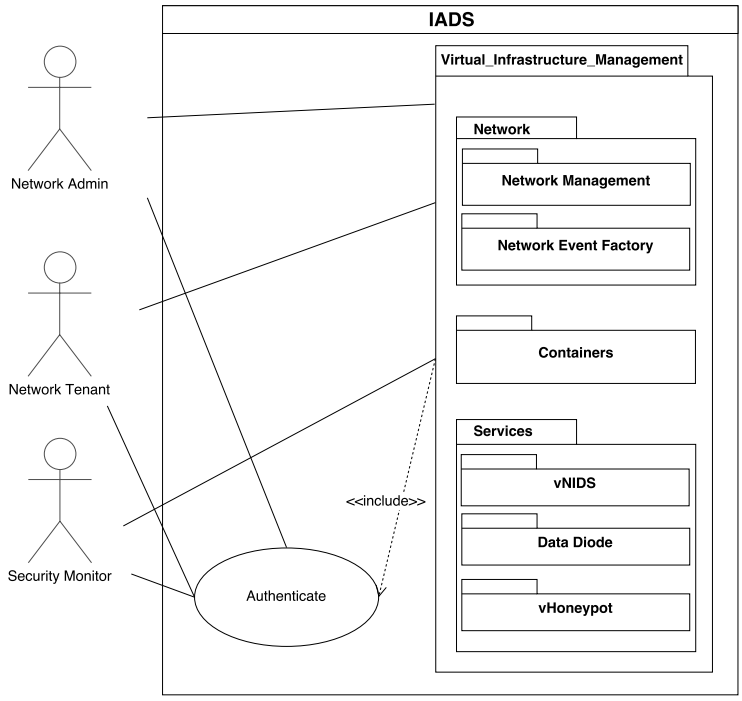


Figure 4.7: The virtual infrastructure management use-case package.

It is important to mention that virtual services (such as IDS and honeypot), in the context of this thesis are perceived as network abstraction service deployments, being totally independent of the respective running software package logic. That being said, an HoneyPot or a vNIDS deployment is an SDN service that allows copying network packets to specific container images and not with the detection of security events within the container itself. For instance, three instances of a vNIDS service can be deployed on the network all of them based on the same container image (e.g. Snort).

Similarly, the Virtual infrastructure monitoring use-case package can be further decomposed into two other packages as shown in Figure 4.8.

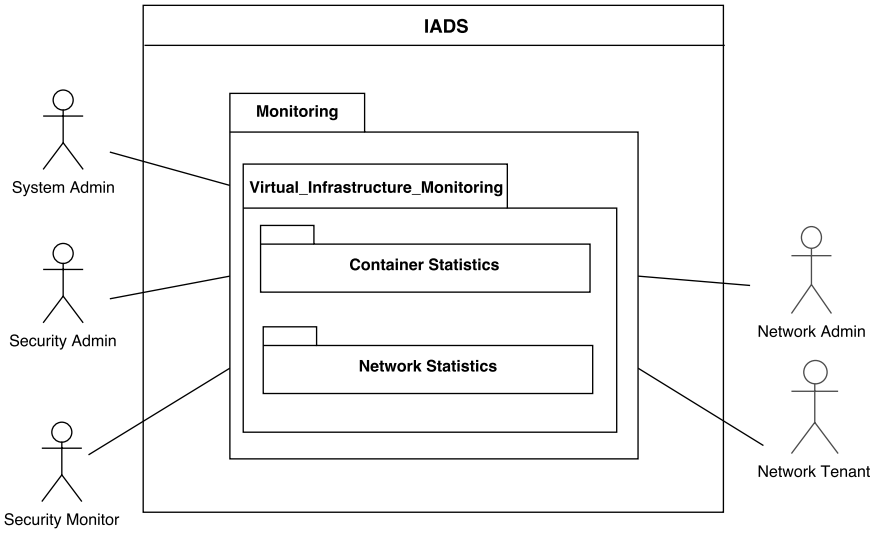


Figure 4.8: The virtual infrastructure monitoring use-case package.

Container Statistics use-case package groups all the use-cases related to the visualization of statistics (CPU, memory, network) and respective live charts related to the container infrastructure. Network Statistics use-case package is related to all the statistical information that can be retrieved from the SDN network by means of the SDN controller.

4.3 Functional requirements

Following the conventions stated in Subsection 4.2.2, the list of elicited functional requirements is listed below. Please recall these requirements were mostly gathered using the use-cases approach (see Subsection 4.2.3). Use-case diagrams per package and their descriptions are presented in Annexes A and B, of Volume II, respectively.

Table 4.5: System functional requirements organized per system package.

ID	Requirement	Priority
UM_FR1	Authenticate – The system has to provide a way for users to be authenticated	MH
UM_FR2	Create Account – The system has to provide a way account creation in order to allow the segregation of functionalities	MH
UM_FR3	Remove Account – The system has to provide a way of removing user accounts	MH
UM_FR4	Modify Account – The system should provide a way of modifying user account information	SH
UM_FR5	List Accounts – The system has to provide a way of listing all the registered users in the system	MH

UM_FR6	View profile – The system should provide a specific view for the profile associated with any of the registered users	SH
CM_FR1	Add physical host to the platform – The system must provide a way to associate physical hosts where containers can be deployed	MH
CM_FR2	Remove physical host from the platform – The system must provide a way of removing a previously added physical host	MH
CM_FR3	List physical hosts – The system must provide a way of listing all the physical hosts associated with the platform	MH
CM_FR4	Associate container image registry – The system must provide a way of associating a container repository to the system where container images /templates can be deployed	MH
CM_FR5	Disassociate container image registry – The system must allow the removal of the information of a container registry previously associated with the platform	MH
CM_FR6	Add container image to registry – The system should provide a way of adding container images/templates to the registry	SH
CM_FR7	Remove container image from registry – The system should provide a way of removing a previously added container image from the registry	SH
CM_FR8	List container images from registry – The system has to provide a way of listing the available container images in the registry	MH
CM_FR9	Start Container – The system has to provide a way of starting a container based on a previously added image	MH
CM_FR10	Stop running Container – The system has to provide a way of stopping a running container	MH
CM_FR11	List Running Containers - They system has to provide a way of listing all the running containers on the SDN network	MH
CM_FR12	List running containers belonging to network tenant – The list of running containers should allow a way of filtered by the network tenant that owns the container running service	SH

CM_FR13	Filter by network service name – The running containers list should allow filtering by the network virtualized service that is running inside the container	SH
CM_FR14	Filter by IP address – The running containers list should be filterable by Ip address	CH
CM_FR15	Filter by Mac Address – The running containers list should be filterable by mac address	CH
CM_FR16	Filter by network service name – The list of images in the registry should be also filterable by network virtualized service	SH
CM_FR17	Attach container to SDN network – The system must provide a way of attaching containers to the underlying SDN controllable network	MH
CM_FR18	Request IP Address on SDN network – The system should provide a way for containers to request an IP address on the SDN network	SH
CM_FR19	Detach container from the SDN network – The system has to have a way of detaching stopped containers from the SDN network	CH
CM_FR20	Pull image from container registry – The system must be able to pull a previously added container image from the registry	MH
NM_FR1	Create Logical Sub-Network – The system must provide a way of creating logical sub-networks within the main SDN network	MH
NM_FR2	Remove Logical Sub-Network – The system must provide a way of removing a previously created logical subsection of the network	MH
NM_FR3	List sub-networks – The system has to provide a way of listing all the logical subsections of the network	MH
NM_FR4	Add Host to Sub-Network – The system must provide a way to add a network host to one created sub-network	MH
NM_FR5	Remove Host from Sub-Network – The system must have a way of removing a host from a sub-network	MH
NM_FR6	Rename Logical Sub-Network – An existing sub-network could be renamed once created	CH
NM_FR7	Associate Sub-Network to Network Tenant – A logical sub-network must be associated with a network tenant	MH

NM_FR8	Disassociate Sub-Network from Network Tenant – The system has to provide a way of disassociating a sub-network from a network tenant if it was previously associated	MH
NM_FR9	View Network Information – The system should provide global information regarding the network usage/assets	MH
NM_FR10	View Sub-Network Information – It should be possible to view the information of the logical sub-networks within the overall network	MH
NM_FR11	View Network Topology Graph – The system should provide a global network topology graph	SH
NM_FR12	Filter Network Topology Graph by Sub-Network – The system should provide a way of filtering the network graph by showing/hiding specific parts of the network	SH
NM_FR13	View Sub-Network Topology Graph - The system should provide a way of showing the topology graph of a sub-network	SH
NM_FR14	List Network Hosts – The system must allow the listing of all network hosts	CH
NM_FR15	List Network Links – The system must allow the listing of all network links	CH
NM_FR16	List Network Devices – The system must allow the listing of all network devices (OpenFlow enabled switches)	CH
NM_FR17	Filter By Host Id – The system could provide ways of filtering a host list by Host Id	CH
NM_FR18	Filter by Device Id – The system could provide ways of filtering the device list by device id	CH
NM_FR19	Filter by Link Id – The system could provide ways of filtering the link list by a link Id	CH
NM_FR20	Filter by Mac Address – The system should provide ways to filter network assets by mac address	CH
NM_FR21	Filter by IP Address - The system should provide ways to filter network assets by IP address	CH
NM_FR22	Ensure communication between the added host and all the hosts on the sub-network – When a network host is added to a sub-network the system has to ensure it can contact other hosts on the same sub- network	MH

NM_FR23	Compute shortest path between host pairs – The system has to include methods for computing the best path between network devices	MH
NM_FR24	Create and install flow rules on the devices in the path between each host-pair – The system has to have the ability to install rules in network devices	MH
NM_FR25	Remove flow rules from the devices in the path between each host pair – The system has to have the ability to remove rules installed on the network devices	MH
NEF_FR1	Add message broker topic URI – The system has to provide a way of defining a broken and topic URI for it to publish network events (to identify the destination domain processor)	MH
NEF_FR2	Remove message broker topic URI – The system has to provide the ability to remove a previously associated domain processor for network event publishing	MH
NEF_FR3	Publish network events – The system has to publish network events to a domain processor	MH
NEF_FR4	Publish device events – The system has to publish device events to a domain processor	MH
NEF_FR5	Publish link events – The system has to publish link events to a domain processor	MH
NEF_FR6	Publish topology events – The system has to publish network topology events to a domain processor	SH
NEF_FR7	Publish host events – The system has to publish network events related to network hosts to a domain processor	MH
NEF_FR8	Publish network controller events – The system has to publish network controller events to the domain processor	CH
NEF_FR9	Stop publishing events to message broker topic – The system has to have methods to allow starting the publication of network events	MH
NEF_FR10	Start publishing events to message broker topic - The system has to have methods to allow stopping the publication of network events	MH
VN_FR1	Enable vNIDS service – The system must provide a way for tenants to enable a vNIDS service	MH
VN_FR2	Disable vNIDS service – The system has to provide a way to disable a running vNIDS service	MH

VN_FR3	Add host to vNIDS service – The system has to allow tenants to add hosts to be monitored by a running vNIDS service	MH
VN_FR4	Remove host from vNIDS Service – The system has to allow tenants to remove hosts that are being monitored by a vNIDS service	MH
VN_FR5	List hosts on vNIDS service – The system has to list all the hosts that are being monitored by a specific vNIDS service belonging to a tenant	MH
VN_FR6	List all vNIDS containers – The system should have a list of running containers associated with the vNIDS application	MH
VN_FR7	List vNIDS containers belonging to tenant – The system should have a way to filter the list of running containers by the vNIDS service and the tenant it belongs	SH
VN_FR8	Configure vNIDS service – The system should let a network tenant configure its own vNIDS service	MH
VN_FR9	Create Scalability Policy – A running vNIDS must have associated a scalability policy	CH
VN_FR10	List scalability policies – The system must allow listing of the scalability policies associated with a given vNIDS service	CH
VN_FR11	Remove scalability policy – The system must provide a way of removing an associated scalability policy from a vNIDS service	CH
VN_FR12	Start vNIDS container – The system must have internal methods to start a vNIDS based container	MH
VN_FR13	Start monitoring host traffic – The system must have internal methods to start monitoring host traffic on a specific vNIDS service	MH
VN_FR14	Compute path between host and container – The system must be able to compute the shortest path between a host being monitored and the respective container running the vNIDS service	MH
VN_FR15	Stop monitoring host traffic – The system must have internal methods to stop monitoring host traffic on a vNIDS service	MH

VN_FR16	Remove copy/forwarding rules from affected devices – The system must allow the removal of rules that provide copies of the traffic of a specific host to vNIDS containers from the devices of the network	MH
VN_FR17	Install rules to copy/forward traffic on all affected devices – The system must be able to install rules to copy traffic generated by network hosts on the network devices of the network	MH
VN_FR18	Stop vNIDS container – The system must have internal methods to stop a running vNIDS container	MH
VN_FR19	Remove all flow rules to copy network traffic from hosts – When stopping a vNIDS container the system should be able to remove all the rules to duplicate traffic that were previously providing copies of the traffic to the stopped container	SH
VN_FR20	Start monitoring service for the vNIDS service – The system should monitor the running vNIDS containers periodically so that it is able to scale the service according to the defined scalability policies	MH
VN_FR21	List all vNIDS services – The system should list all the running vNIDS services (i.e. ability to differentiate vNIDS belonging to different network tenants)	SH
VN_FR22	Stop monitoring service for the vNIDS service – When a vNIDS container is stopped the monitoring service for the scalability of that container should be stopped as well	MH
VH_FR1	Deploy vHoneypot – The system must provide ways for tenants to deploy virtual honeypots on their sub-network	MH
VH_FR2	Remove vHoneypot – The system must provide a way of removing a previous deployed honeypot	MH
VH_FR3	List all vHoneypots – The system must provide the ability to list all the vHoneypots on the global network	MH
VH_FR4	List vHoneypots belonging to Tenant – The system must provide the ability to filter the list of network vHoneypots to display only those belonging to a specific network tenant	MH
VH_FR5	Stop vHoneypot container – The system must internally be able to stop a vHoneypot container	MH
VH_FR6	Remove flow rules from all the devices – The system must remove rules from network devices that were previously redirecting traffic to a deployed vHoneypot	MH

VH_FR7	Compute the shortest network paths between all devices and the vHoneypot container – The system must be able to compute the shortest path between a vHoneypot container and an attacker on the network	MH
VH_FR8	Redirect traffic with SRC or DST to the given IP address range to the vHoneypot container – The system has to provide a way for tenants to configure the IP address range that a vHoneypot should be targeting	MH
VH_FR9	Install flow rules to forward the traffic to the vHoneypot on all network devices – The system must be able to install rules on the network devices so that any traffic generated with source or destination to the IP range defined when the vHoneypot was deployed is redirected to the vHoneypot container	MH
VH_FR10	Start vHoneypot container – The system must internally be able to start a vHoneypot container	MH
VH_FR11	Define IP address range for vHoneypot – The system must provide a way for tenants to define a IP address range for the vHoneypot operation	MH
DD_FR1	Set Network Link as a data diode – The system has to allow tenants to set a network link as a unidirectional gateway/data diode	MH
DD_FR2	Set Network Link as a regular link – A previously link set as a data diode must be removable by the network tenant	MH
DD_FR3	List all network links set as a data diode – The system should allow the listing of all data diodes on the network	SH
DD_FR4	Filter network link by sub-network name – The list of data diodes should be filterable by network name	CH
DD_FR5	Filter network link by network tenant – The list of data diodes must be filterable by the network tenant username	CH
DD_FR6	List all network links set as a data diode for links belonging to a tenant sub-network – The system must provide a way of listing data diodes that are deployed by a tenant	MH
DD_FR7	Remove rules associated with the respective data diode – The system should allow the removal of flow rules that block traffic on a link set as a data diode	MH

DD_FR8	Install rules to drop the network packages on the network device that contains the edge link – The system should be able to install rules on devices so that traffic on a link set as a data diode only occurs on the opposite direction of the one specified for the data diode operation	MH
NS_FR1	Network Statistics – The system must consider the display of network statistics	MH
NS_FR2	View host statistics – The system must display network statistics related to network hosts	CH
NS_FR3	View device statistics – The system must display network statistics related to network devices	CH
NS_FR4	View link statistics – The system must display network statistics related to network links	CH
NS_FR5	Real time plots – Network statistics should be visible in the form of dashboards/real time plots	SH
CS_FR1	Container real-time statistics – The system should display real time statistics for running containers (those associated to network services such as vNIDS and vHoneypot)	MH
CS_FR2	View memory consumption – The system should display statistics related to the memory utilisation of a specific container	MH
CS_FR3	View host CPU usage – The system should display metrics related to host CPU usage of a running container	MH
CS_FR4	View Network bandwidth usage – The system should display metrics related to the network bandwidth usage of a given running container	MH

4.4 Non-functional requirements

Following the approach detailed in section 4.2.3 all the elicited non-functional requirements are presented in respect to the specific system quality attribute and system scope. There are five sub-categories for non-functional requirements:

- Security Requirements,
- Performance Requirements,
- Availability Requirements,
- Operational and Environmental Requirements,
- Interoperability Requirements.

Below presented requirements, despite being named IADS, also apply to the SDN scope of the IADS platform. Requirements that do not apply were removed from this report. As a result, in some tables requirement identifiers do not follow a sequential order.

4.4.1 Security requirements

Table 4.6 details the security requirements in the context of the SDN scope of the IADS platform.

Table 4.6: Security non-functional requirements of the system.

ID	Requirement	Priority
IADS_NFR1	The system must not store user passwords in plain-text. All sensitive information must be stored encrypted using a robust hashing algorithm	MH
IADS_NFR2	Network tenants must have access to the platform web interface but not to the network controller (network operating system) interface	MH
IADS_NFR3	Hosts belonging to two different logical sub-networks must not be able to communicate with each other	MH
IADS_NFR4	Communication between the network devices (OpenFlow enabled switches) and the network controlled has to be encrypted	SH
IADS_NFR5	The network controller nodes and the hosts of a logical tenant sub- network should be in a different physical network so that the tenant hosts cannot establish a connection with the controller nodes	MH
IADS_NFR6	In case a network link is set as a data diode in a given direction, network packets must not be able to flow in the specified direction (only in the opposite direction)	MH
IADS_NFR7	The system must not allow, when possible, sensitive information to be transmitted between components without encryption.	MH
IADS_NFR10	Management tasks should, ideally, be transmitted in off-band channels, and using secure communications.	CH
IADS_NFR11	All accesses should be registered, using log mechanisms, for non- repudiation effects	MH

4.4.2 Performance requirements

Table 4.7 lists the performance requirements elicited for the SDN scope of the IADS platform.

Table 4.7: Performance non-functional requirements.

ID	Requirement	Priority
IADS_NFR12	Software defined networking core logic must be implemented at the application level of the network controller and not rely on high- latency controller external API's (e.g., not use the controller REST API)	SH
IADS_NFR13	The network controller should be able to balance the master ships of the associated OpenFlow enabled switches across different controller nodes in order to maximize the overall system performance	MH
IADS_NFR14	The network controller software should be selected according to performance metrics	MH

4.4.3 Availability requirements

Availability requirements are listed in Table 4.8.

Table 4.8: Availability non-functional requirements.

ID	Requirement	Priority
IADS_NFR16	The network controller should be distributed to avoid single point of failures	MH
IADS_NFR17	Every network device (OpenFlow enabled switch) must be associated with different network controller nodes in a master-slave fashion so that its availability is not compromised by a controller node failure	MH
IADS_NFR18	Network services that rely on scalability policies (e.g., vNIDS) should be continuous monitored to be automatically scaled if the defined policy is reached	MH
IADS_NFR19	The platform components should be fault tolerant.	CH
IADS_NFR20	The transport and processing mechanisms of the platform should be redundant.	SH
IADS_NFR21	For SDN-based communication, proactive (pre-set) flows should be privileged over reactive (new) flows.	MH

4.4.4 Operational and environmental requirements

Elicited non-functional requirements belonging to Operational and environmental category are listed in Table 4.9.

Table 4.9: Operatonal and environmental non-functional requirements.

ID	Requirement	Priority
IADS_NFR24	Components should be containerized as much as possible, and be isolated to allow a neutral deployment.	SH
IADS_NFR25	The deployment of the platform should not impose, when possible, constrains on the availability of the IACS infrastructure.	MH

4.4.5 Interoperability requirements

Interoperability non-functional requirements are summarized in Table 4.10.

Table 4.10: Interoperability non-functional requirements.

ID	Requirement	Priority
IADS_NFR26	Logs must be based on a format that assures interoperability between the logs of each component.	SH
IADS_NFR28	The platform should be able to have new components integrated into it (such as new probes).	MH

4.5 Design constraints

The system also has a set of design constraints that inherently affects its development. Design constraints are summarized in Table 4.11.

Table 4.11: System design constraints.

ID	Requirement	Priority
IADS_DC1	The system must have a web interface	MH
IADS_DC2	All the events transiting within the IADS system must be encoded following the data model established for the platform. The data model definition is not part of this thesis but can be consulted in AtenaConsortium4.3 2017	MH

IADS_DC4	All the network devices on the SDN scope must support the OpenFlow protocol (versions 1.0 – 1.3)	MH
IADS_DC7	The container infrastructure must contain an agent to attach containers to the SDN network	MH
IADS_DC8	A virtual probe container (IDS or honeypot) must have a pre-defined name on its SDN virtual Ethernet interface	MH
IADS_DC9	A virtual probe to be attached to the SDN network must be tagged in order for the agent to identify the container before attaching it to the SDN network	MH

4.6 Chapter wrap-up

By adopting an engineering approach to the requirements elicitation process, the team developing the IADS was able to better understand and grasp the complexity of the overall system. The stratification of the IADS architecture in functional blocks and the prioritization of requirements allowed for a clear segmentation of the platform into manageable subsystems and components – reducing the risk for project failure. Identifying the platform actors and looking at platform from the perspective of the operations each specific role was allowed to perform, was a key strategy which encouraged the development team to engage in the collective design effort of the main system functionalities. Furthermore, the development of several prototypes/tests along the elicitation process helped finding requirements (and design constraints) that would have been really hard to find at later stages of the project and that likely would require a complete revision of the requirements document. Globally, we think the requirements elicitation process is one of the most important parts of software development – affecting its overall lifecycle. The presented requirements, and the level of detail that was devoted to the section, is further reflected on the completeness of proposed architecture.
The elicited requirements were formalized in the form of use-cases. Use-case diagrams and use-case descriptions are a substantial part of this thesis' annexes (Vol. II).

Chapter 5

Software Architecture

This chapter details the software architecture of the platform proposed in this thesis. Since the architecture highly depends (and inherits) from the architecture of the adopted network controller, Section 5.1 provides an overview of distributed controller architectures. Within this section, the architectures and application development models of the most known distributed controllers are explored (OpenDaylight and ONOS - Subsections 5.1.1 and 5.1.2, respectively). Performance comparison between both controllers are provided in Subsection 5.1.3. The architectural differences, programming model and their performance aspects lead to the selection of one controller in Subsection 5.1.4. Finally, in section 5.2 an architecture overview of the proposed subsystem is provided. High-level architecture is presented in Section 5.3, whilst specific application and component architectures are further detailed in Section 5.4. Section 5.5 concludes the chapter.

5.1 Distributed controller architectures

In SDN the network controller represents the logically centralized control plane of the network which can access all the network switches in the underlying network. The fact it is logically centralized does not mean it is only composed by a single controller node. In critical infrastructures, availability is the most important design goal. If a network controller is represented in the architecture by a single node, it can easily be seen as a single point of failure. There are alternative architectures in which the control plane is also physically distributed despite being logically centralized.

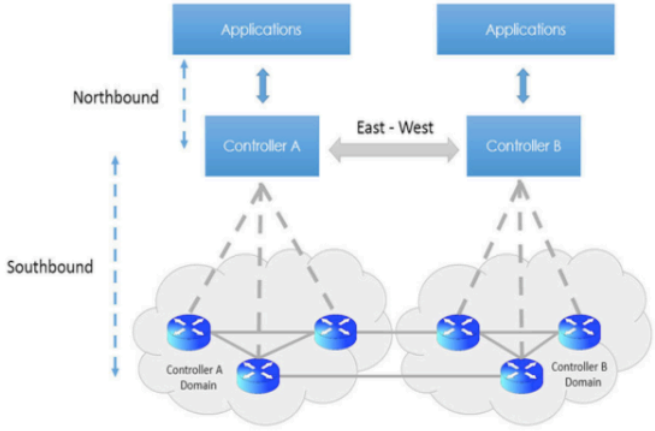


Figure 5.1: Distributed controller architectures (Naseer 2016).

In those distributed architectures, controller nodes share the network state (through messages) via their east/westbound interfaces and organize switch mastership among nodes (switches are configured to have one node as master and other nodes as slaves). Distributed controller architectures (Figure 5.1) bring numerous advantages when compared to single node architectures: resilience of the control plane is improved and the control overhead is reduced.
Typically, distributed network controllers employ strong consistency models, meaning data is not made available to the switches unless the controller has been fully updated. This type of model is used when exchanged information is critical and outdated information can create anomalies in network operation. The RAFT algorithm (Figure 5.2) is the base for strong consistency models in SDN controller clusters. RAFT consists of a cluster of nodes, each having a log to maintain. This log is fully replicated in all nodes achieving consensus through replicated state machines. These state machines process identical sequences of commands to produce the same output. In order to keep consistency, elections occur between nodes – a distinguished leader (responsible for replicating the log to other nodes) is elected within the cluster. Other nodes request the network state from the leader, which serves and guides them through log changes to avoid inconsistencies. In case of a leader failure, any other node is eligible to become leader. The leader selection is achieved through elections between nodes in randomized time intervals. All nodes in the cluster move to candidate state and vote for themselves as leaders. As the election time is randomized, the likelihood of having split-votes is quite low. In case of a split vote situation, a new election takes place. When a node is elected as leader, all the others move to follower state (Lamport et al. 2014; Zhang et al. 2017).

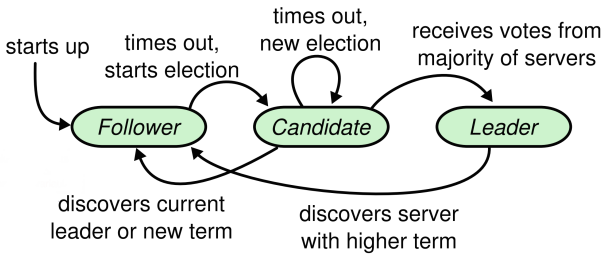


Figure 5.2: The RAFT algorithm in SDN distributed controllers (adapted from Zhang et al. 2017).

Although there are multiple network controller software packages, only a small minority is distributed. The two main distributed controllers are OpenDaylight and ONOS. An architectural (and application development model) comparison between both is provided next.

5.1.1 OpenDayLight

OpenDaylight is an open source project formed and hosted under the Linux Foundation with the goal of furthering the adoption and innovation of SDN. OpenDayLight aims at building a common SDN stack and enforcing programming standards so each of the common network functions and services can be developed on a collaborative effort by different industry members (Cisco 2013). Founded on April 2013, it was first released in February 2014 (Hernandez 2016). Since then, eight stable releases were issued at a constant period of around 6 months. The current release (codename Oxygen) is available since March 2018.

OpenDaylight is licensed under the Eclipse Public License v.1.0 (ODL-license 2017). Initial OpenDaylight consortium members included a broad range of industry players, from leading networking vendors such as Cisco and Brocade to IBM, HP and Microsoft. While the project has increased to 49 partners, some key players have been abandoning the project or reducing the investment. Examples are Microsoft and Citrix (nowadays only silver project members) and IBM and VMWare (no longer project members). If a single reason has to be pointed out to explain OpenDaylight's success and adoption it was for sure the commitment of its several members. In fact, as a result of the partner's commitment, a dedicated official security-response team exists in order to apply a formal internal process to handle security related patches when a serious vulnerability is disclosed (Thenewstack 2015).

OpenDaylight has started as a non-distributed SDN controller, as an effort for multiple players to have a common SDN framework which promoted openness, transparency and agility for the development of SDN solutions. Its architecture have been evolving to support controller clustering and improve the degree of abstraction. For this reason, architecture wise the controller has suffered deep modifications which negatively impact performance, compatibility and the quality of available code and documentation. The controller is written in the Java Programming Language and leverages the OSGi framework – which allows plugins and applications to be deployed as application containers and registered into the controller during runtime. OSGi also makes it possible to link plugins with the corresponding southbound interfaces. In OpenDaylight, Apache Karaf is used as the container management platform. Opendaylight's architecture (Figure 5.3) can be sliced into three main logical layers, although the architecture can also be viewed as an horizontal service oriented architecture in which all the micro-services communicate through a centralized bus (the Model Driven Service Layer Abstraction – MD-SAL).

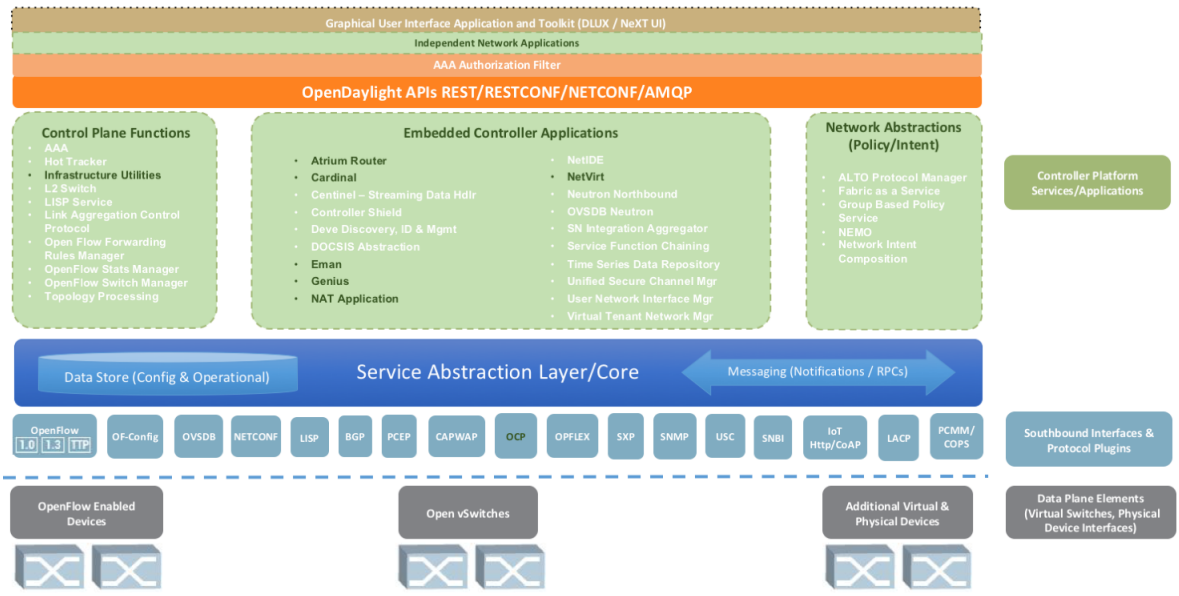


Figure 5.3: OpendayLight controller architecture (from OpenDayLight 2018).

The fundamental layers of the Opendaylight controller architecture are:

- **Top-Layer/ Northbound interface** – This layer provides external access for applications running a different namespace to access the control layer (middle services). The access can be achieved through standard interfaces such as REST API's, RESTCONF

(YANG models to REST), NETCONF or AMQP. All those protocols are enabled as external installable plugins in the controller which directly communicate with the service abstraction layer. For instance, the AMQP plugin exposes to the northbound interface the datastore, notifications and RPC registries of MD-SAL. This protocol can then be used to communicate with a broker either loaded in Karaf itself or an external broker such as ActiveMQ.(OpenDayLight-Wiki 2018) OpendayLight Graphical User Interface Applications (DLUX), communicate with the controller via the REST API(OpenDayLight-Tutorial 2016).

- **Middle Layer/Controller Layer** - The layer in which the controller communicates with the underlying network infrastructure with the help of control plane functions and embedded controller applications. All those are OpenDaylight extensions that are loaded in Karaf and use MD-SAL to communicate with the southbound plugins. In fact, MD-SAL acts as an active registry for brokering contracts between service providers (control plane functions and protocol plugins) and the consumers – the applications. In the middle layer, several network services/functions are part of the shipped base. These include services for topology discovery, a forwarding manager for forwarding rules management, a switch manager to identifying networking elements in the underlying physical topology:

  1. Topology Processing – is responsible for discovering the OpenFlow topology using LLDP and putting them into the operational data store for applications' use.
  2. OpenFlow Stats Manager – Network service for managing statistics and counters across OpenFlow enabled nodes: flows, queues and groups. Implements the collection of statistics by sending requests for statistics to all active nodes (i.e. the managed switches) of the intelligent network and stores their responses to operational datastore (Paliou 2016).
  3. OpenFlow Forwarding rules manager – A network service for registering and obtaining flows from the OpenDaylights datastore. It manages key forwarding rules, resolve any conflicts between those rules and validates them (Paliou 2016).
  4. OpenFlow Switch Manager - Provides information for nodes and the ports which are connected. As new network elements are discovered, their information is stored in the Switch Manager data tree.

- **Southbound interfaces** - Opendaylight supports the Openflow protocol between versions 1.0-1.3. Appart from Openflow, a multitude of other protocols are also supported at the southbound interface. Examples are CoAP, NetConf, HTTP, SNMP, BGP, among others.

Figure 5.4 shows how the different control plane functions co-exist and interact both with each other and with southbound providers.

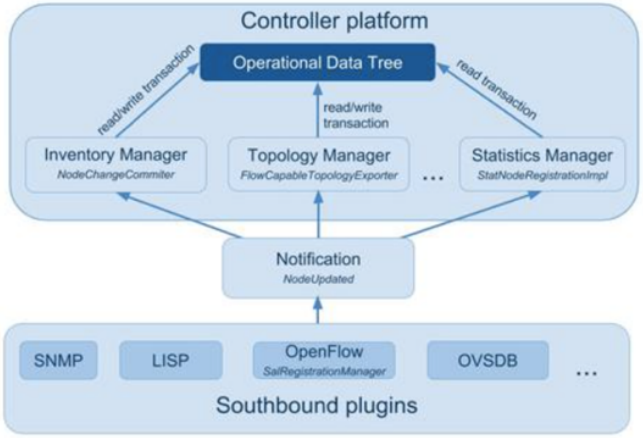


Figure 5.4: OpenDaylight network function interaction (from Paliou 2016).

When an OpenFlow enabled switch first tries to connect with the controller, a notification is issued to either the Switch Manager, the Topology Processing Service and the OpenFlow Stats manager. All the services share a distributed operational data store built upon the Akka framework (ODLWiki-clustering 2017) although separated between applications in individual data trees. Upon receiving the notification, each network function updates its respective data tree in the distributed store (Mirantis 2015).

The Model Driven Service Abstraction Layer (MD-SAL) is the central piece that enables plugin and application development in OpenDaylight. It goes a step forward from the predecessor (AD-SAL - Application Driven Service Abstraction Layer – deprecated since Hydrogen) by allowing OpenDaylight applications to be designed using the MVC design pattern, as application silos. MD-SAL logically separates plugins, gluing them horizontally as providers and consumers. Models are constructed in the form of YANG definitions, used to define the plugin data model, the services it exposes to other plugins or the services it requires from MD-SAL. These services consist of both:

- Remote Procedure Calls – input/output calls.
- Asynchronous notifications – to any registered listeners.

MD-SAL is a reactive architectural measure from the OpenDaylight community to solve the problem of having different teams working on the core platform. With MD-SAL, some API standardization can be achieved while minimizing the chance of having code "honeypots" (big chunks of code) where multiple teams have to touch to implement a given feature. The service abstraction layer manages the contracts and state exchanges between every application by keeping a centralized state.

YANG models are compiled to generate uniform APIs for consumers. Such a design pattern allows dynamic late binding, runtime and compile code generation (LinuxFoundation 2014). Furthermore, plugins are loaded into the controller as soon and they are pushed into the Karaf container platform in the form of OSGi bundles. As a result, the functionality of the controller can be dynamically changed without requiring a reboot. OSGi bundles are built from Maven Build Tools, which handles plugin external dependencies – helping reducing the plugin size. Figure 5.5 illustrates the process of creating an OpenDaylight application from the YANG Model definition until the deployment in the controller.

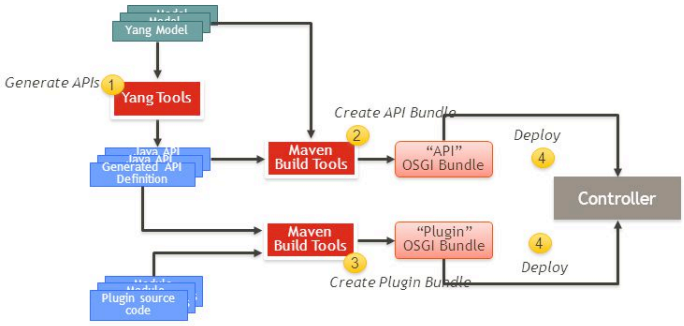


Figure 5.5: OpenDaylight plugin generation from YANG models (from LinuxFoundation 2014).

It is important to mention that the generated APIs are both for the Northbound interface (REST APIs through RESTCONF) as well as standard Java Interfaces to enforce a consistent programming model. At the core of the MD-SAL platform there is a logically centralized data store that keeps relevant state in two different buckets (Seetharaman 2015):

1. **Configuration data store** – always kept persistent (and exposed through RESTConf).
2. **Operational data store** – used for transient data.

All the data referent to configuration instructions (e.g. VLAN mappings to port numbers, flow definitions) are stored in the configuration data store. Other data, such as the system status or statistics are stored on the operational data store (ODL-team 2017). The operational data store is only created when the plugin is loaded into the controller. Opendaylight uses a strong consistency model in its distributed store. The RAFT algorithm is used to keep the network state consistent between controller nodes. The MD-SAL subsystem is responsible for accessing the shared data store to provide data storage for plugins and/or to provide request routing (mappings between a consumer and a provider) through their RPCs or notification services. Figure 5.6 shows how SDN applications are viewed as applicational silos within the network controller.

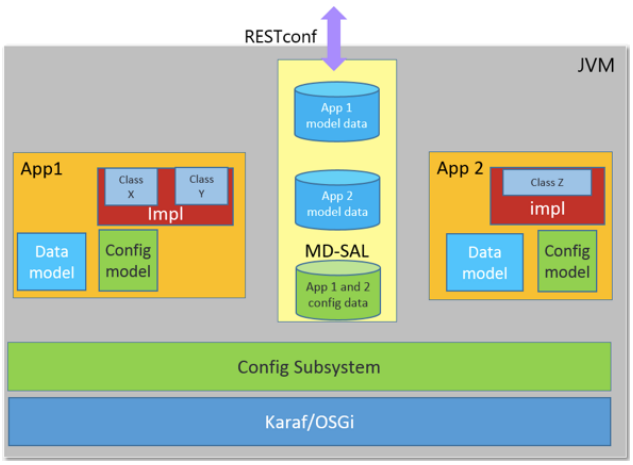


Figure 5.6: OpenDaylight application containers (adapted from Seetharaman 2015).

The MD-SAL subsystem is agnostic to the location of the application. Plugin interfaces are generated from YANG models and built around MD-SAL either if they are a Southbound plugin or if they are a plugin supposed to be accessed though the Northbound interface. MD-SAL only cares about the notion of provider and consumer and ultimately can be seen only as a "request routing" service (Thenewstack 2015).

As an example, if we think about a simple network application such as a packet processor application, a YANG model has to be defined to specify which services from MD-SAL the application requires. In this case those would be:

- mdsal:binding-async-data-broker (to write to the data store).
- mdsal:binding-notification-service (to access notifications).
- mdsal:binding-rpc-registry (to access the rpc registry).

The YANG model would also contain all the services the application exposes and its respective data model. Application Java interfaces would be generated afterwards using YANG Tools as well as all the REST endpoints to externally access the application services. The programmer would then need to implement those interfaces and request specific services from the md-sal rpc registry. An example would be to create the necessary bindings for the *PacketProcessingService* which is registered in the MD-SAL by the controller (to react to packet_in events received by the notification service). Furthermore, in some cases it is necessary to access the operational store of other applications or to react to events on those stores. This kind of programming philosophy despite helping to maintain a consistent interface and to reduce the need for big core modules has the disadvantage of increasing the learning curve for application development. The programmer has to be aware of the several OpenDaylight plugins (some are not even enabled by default), their API's and their respective data model. Furthermore, since the main application codebase depends on generated interfaces, the programmer has to generate a new model (and compile it) in order to generate new interfaces it might want to implement. All this effort just to create a few more methods on the application.

5.1.2 ONOS - Open Network Operating System

Unlike OpenDaylight which started as a single-node controller, the Open Network Operating System was the result of a series of prototypes developed by On.Lab to accomplish a distributed controller with specific quality attributes in mind: availability, modularity and performance. In the first paper presenting the ONOS framework, the developers present the main requirements leading to the controller development (Berde et al. 2014):

1. High Throughput: up to 1M requests/second.
2. Low Latency: 10 - 100 ms event processing.
3. Global Network State Size: up to 1TB of data.
4. High Availability: 99.99% service availability.

Started in 2014 as an Apache 2.0 licensed project, ONOS has seen more than 14 releases and have seen the number of project members increase on a regular basis. The Open-Source project is now supported by companies such as Google, Samsung, AT&T, Ericsson, Cisco, Huawei or T-mobile (ONOS-website 2018). Since 2006, the project is also under the

umbrella of the Linux Foundation (ONOSproject 2016). Despite the high number of releases, the ONOS API for SDN application developers have been quite stable overtime. Changes to the project have been related to the internal architecture, to improve the controller resilience and performance while keeping external interfaces (and programming model) unchanged. As an example, ONOS started by using Cassandra as the chosen distributed store and Zookeeper for cluster coordination. However, On.Lab has realised the kind of data SDN applications needed to store was so simple that such a complex database model like Cassandra was not justified. Hence, ONOS replaced Cassandra by Hazelcast - an high-density in-memory map data store. As an effort to bring the storage model more close to the program model, ONOS uses nowadays the MapDB database for its distributed store. MapDB provides Java Maps, Sets, Lists, Queues and other collections backed by off-heap or on-disk storage (MapDB 2018). Clustering mechanisms also have changed over time. Due to the deprecation of the Zookeeper, recent versions of ONOS rely on the Atomix framework for cluster coordination. This change greatly contributed to improve the controller performance since the cluster coordination can now leverage MapDB to achieve different distributed primitives, depending on the criticality of the data to be stored. This is made possible by having each service's store implement the appropriate distribution mechanism. SDN developers can now choose between two consistency models: strong and eventually consistent stores (ONOS-wiki1 2018):

1. **Eventually Consistent** - Backed by data structures such as *EventuallyConsistentMap*, this consistency model fully replicates all its state between all nodes in the cluster. This means each node in the cluster will have a full copy of the map contents stored in memory and data will not survive a node reboot. This consistency model provides weaker consistency guarantees in favor of superior read and write performance. Applications can also configure eventually consistent maps with a `ClockService` to ensure each map replica applies changes to local state in the correct order. The model uses a lightweight background process known as anti-entropy (also known as the optimistic Gossip replication algorithm) to ensure all replicas eventually converge to the same state.
2. **Consistent** - Data structures (consistent maps) backed by the RAFT consensus algorithm. This model ensures that any changes made to the a given key on the distributed map are serialized to disk. Furthermore, the entire map space is partitioned between all the nodes in the cluster. For instance, in a 3 node cluster each consistent map is partitioned into three shards and each shard is distributed between the three nodes. In case of a node partition, another cluster node will have all the data of the "lost" shard.

Figure 5.7 shows data replication between onos nodes. Internally, ONOS uses eventually consistent stores for Device, List and Host management. Each of these "subsystems" stores are completely independent from each other. The device mastership (the associating between OpenFlow switches and the master controller node) is assured by a consistent model. All controller nodes start by being in the *STAND_BY* mode when a switch attempts to connect. Elections then start to take place in a timely manner (however randomized) in order to elect the master of the switch (changing one of the nodes state to MASTER). Other critical information, such as the FlowRuleStore, also employ a strong consistency model. Despite the internal choices, SDN developers are, however, free to adopt any of the two types of consistency models in their applications (ONOS-wiki2 2018).

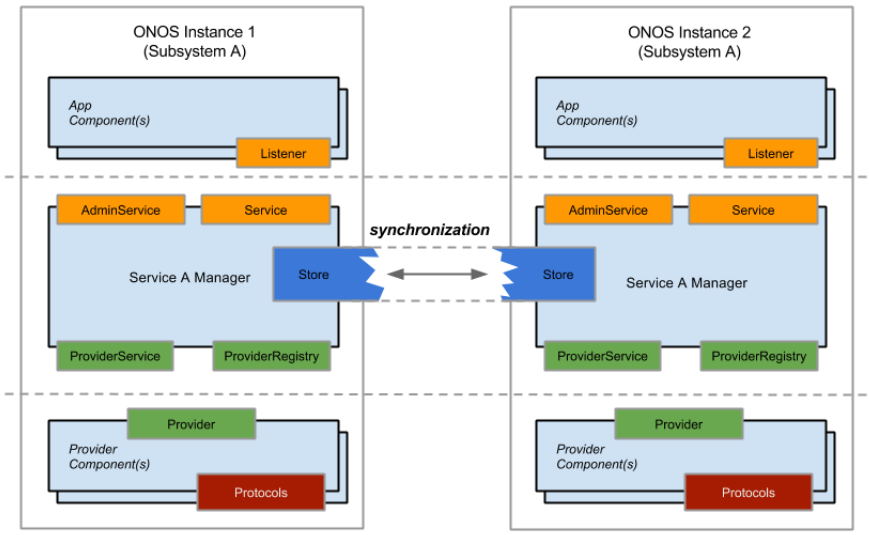


Figure 5.7: ONOS cluster synchronization (from ONOS-wiki2 2018).

Similarly to OpenDaylight, ONOS also uses an OSGi runtime framework to promote modularity and flexibility in application development, using Apache Karaf as the OSGi runtime environment. However, it differs from OpenDaylight since applications do not follow an Model-view-controller (MVC) design pattern. ONOS leverages the Apache Felix framework to create application bundles (oar - ONOS application repository files) and to provide late-binding (at runtime) for OSGi services and components (through `@Service` and `@Component` annotations) – core service dependency injection (ONOS-wiki4 2018). Unlike OpenDaylight, the ONOS architecture (Figure 5.8) is not a service oriented architecture (SOA) and is much better interpreted as a layered design. In fact, some authors state ONOS application development model is similar to the one employed by OpenDaylight before MD-SAL (Goransson et al. 2014).

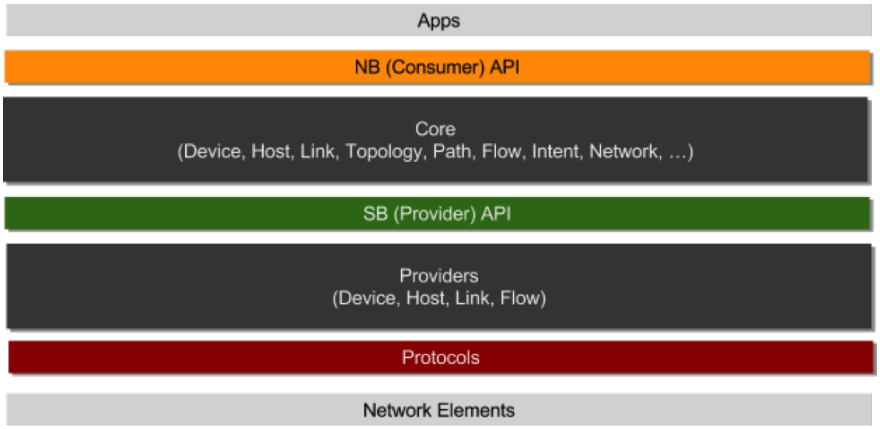


Figure 5.8: ONOS stack architecture (from ONOS-wiki3 2018).

The ONOS architecture is based on the notion of *Providers* and *Consumers*. Providers and consumers glue together to create what ONOS developers call ONOS subsystems - a vertical slice on the architectural layer stack. ONOS is composed of the following subsystems (ONOS-wiki3 2018):

- **Device Subsystem** - Manages the inventory of infrastructure devices (switches).
- **Link Subsystem** - Manages the inventory of infrastructure links.
- **Host Subsystem** - Manages the inventory of hosts and their location in the switch fabric.
- **Topology Subsystem** - Manages net topology graph representations.
- **PathService** - Computes/finds paths between infrastructure devices or between end-station hosts using the most recent topology graph snapshot.
- **FlowRule Subsystem** - Manages inventory of the match/action flow rules installed on infrastructure devices and flow statistics.
- **Packet Subsystem** - Allows applications to listen for data packets received from network devices and to emit data packets onto the network via device ports.

One of the most interesting aspects of the ONOS subsystems is the fact that each of them provides object abstractions. For example, for an SDN application to install a flow rule it just has to inject the `FlowRuleService` as a dependency (through a Java annotation) prior to the application activation method. The OSGi runtime will provide late-binding making the service available as a Java object when the application is activated. The `FlowRuleService` contains methods to create a `FlowRule` object and to install it on a `Device` object. `Device` objects are obtained through a search method in `DeviceService` or through iteration on the `Topology` object provided by the `TopologyService`. This level of abstraction (and ONOS exceptionally good documentation) results in a much more familiar environment for software developers. Figure 5.10 shows available abstractions provided by the ONOS core.

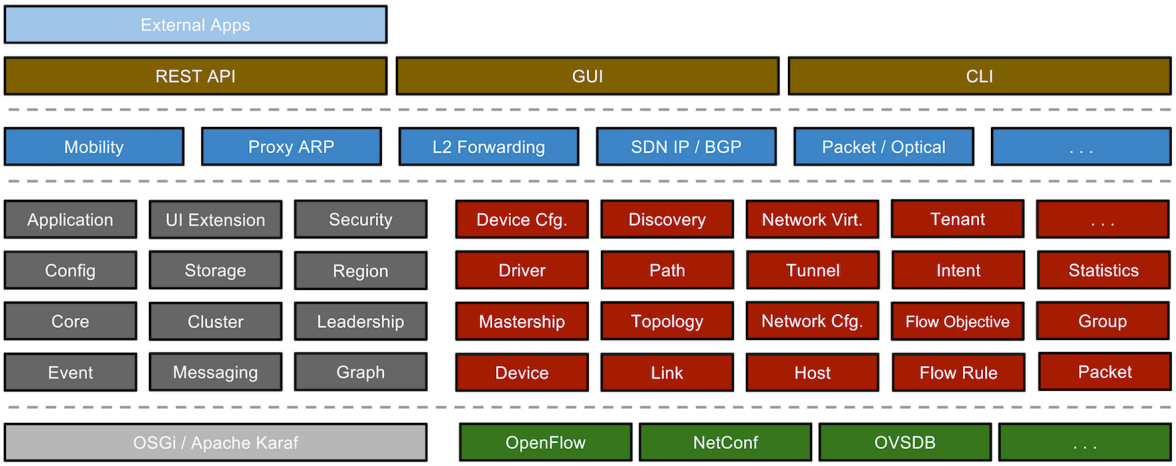


Figure 5.9: ONOS subsystems and abstractions (from ONOS-wiki3 2018).

The ONOS architecture clearly separates the boundaries between the southbound interfaces and the northbound interfaces accessible for applications (dashed lines in Figure 5.10). This separation is ensured by the existence of a "monolithic core" which contains the fundamental subsystems every application requires. As a result, the problems with code honeypots that affected OpenDaylight development (see Section 5.1.1) did not express in ONOS. Although multiple project partners also work in ONOS development, it happens a layer above of the controller core services through applications that extend the controller. CORD, is an

example of a framework where multiple companies work together to create a applications (installed in ONOS) for datacenter use-cases (OpenCORD 2018).

The abstraction between the forwarding plane and ONOS applications uses a publish-subscribe design pattern, heavily supported by events and provider and consumer components (see Figure 5.10).

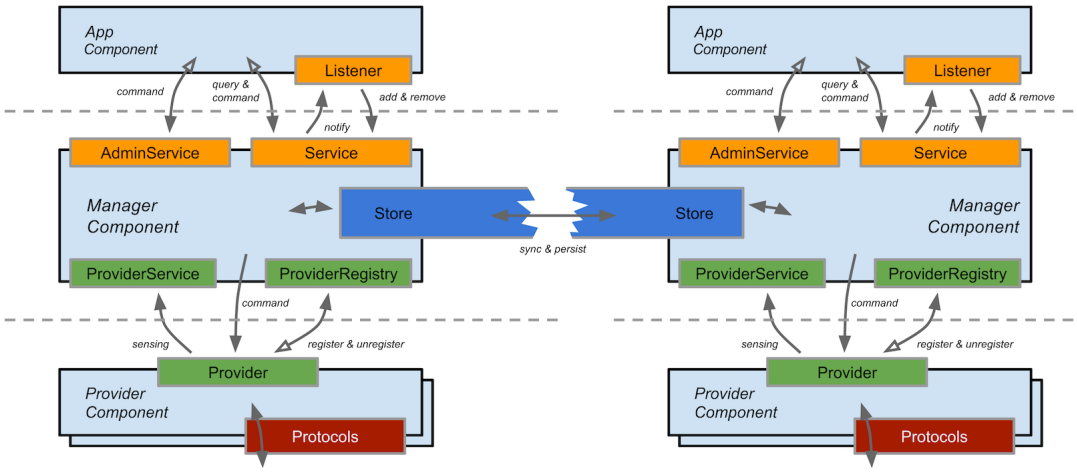


Figure 5.10: ONOS applications and core services relationship (from ONOS-wiki3 2018).

A provider in ONOS is an OSGi component responsible for interfacing with network equipment via protocol specific libraries (e.g. OpenFlow, NetConf, OVSDB). Providers register in the ONOS core via a service manager. The service manager is responsible for interacting with the provider either synchronously (via query-response) or asynchronous by implementing listeners for protocol specific events. The service manager is also responsible for translating protocol descriptions and events into high-level abstractions, homogeneously across protocols. Descriptions and events are immutable objects that map each specific protocol event and description into high-level objects such as `Device`, `Host`, `Flow` or `FlowStatistics`. These objects are then exposed to high-level applications via the northbound interface via a common API. Application components can then query each subsystem or subscribe to network events. Aditionally they can also extend the controller external interfaces: REST, command-line, Web-sockets and GUI.

ONOS also stands out due to the internal concept of Intent-based networking (ONOS-wiki5 2018). Intent-based networking treats network operations in the form of policy and goal, rather than a mechanism. An intent specification (e.g. "ensure connectivity between two hosts") is installed in ONOS (via the intent framework), which compiles the intent and translates it into essential operations in the network environment. The controller internally tracks the network state to ensure the intent is always guaranteed. For instance, if an host-to-host connectivity intent is installed and a switch becomes unavailable, the intent framework will try to find a redundant path to ensure connectivity and automatically create new rules. Figure 5.11 shows intent compilation in ONOS.

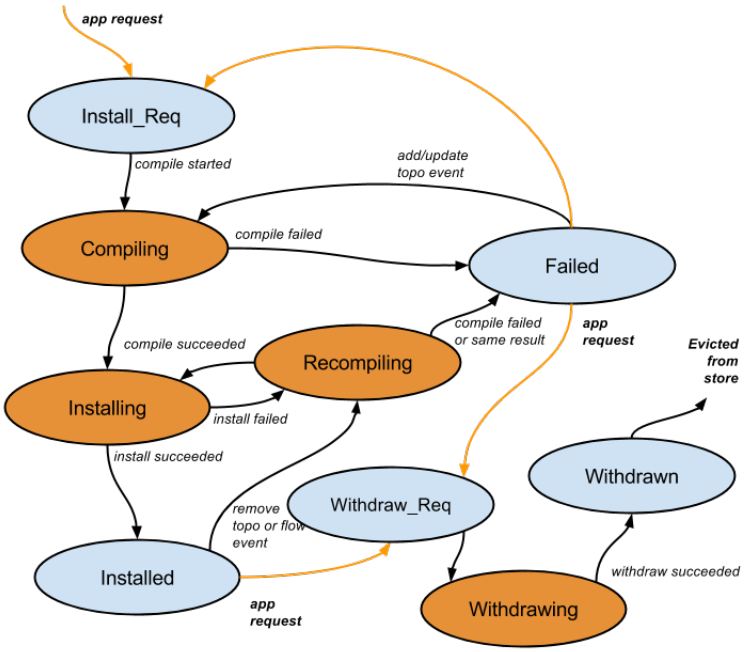


Figure 5.11: ONOS intent framework (from ONOS-wiki5 2018).

As of the last version, the Open Network Operating System supports the following intents (ONOS-JAVAdocs 2018):

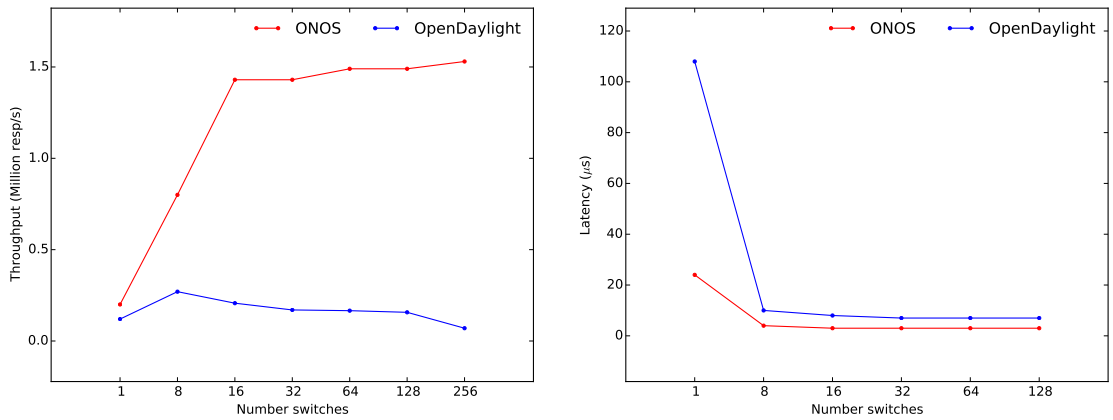
- **Host-To-Host intent**: Ensures connectivity between two hosts in the network.
- **Flow-Rule-Intent**: Groups several FlowRule objects into a single intent.
- **Multi-Point-To-Single-Point-Intent**: Ensures bi-directional connectivity between a group of multiple hosts and a single host (e.g. webserver).
- **Point-To-Point-Intent**: Ensures the connectivity between two points in the network always happen between a pre-defined set of network switches.

In what concerns performance, a single ONOS instance can install just over 700K local flow setups per second. An ONOS cluster of seven can handle 3 million local, and 2 million multi-region flow setups per second (On.Lab 2017).

5.1.3 Performance of Distributed controller

Throughout the years, multiple authors have been evaluating and publishing SDN controller benchmarks in the literature. Many of those works contemplate no-longer maintained controllers (Tootoonchian et al. 2012, Shah et al. 2013, Shalimov et al. 2013, Fernandez 2013 and Y. Zhao et al. 2016) or provide performance metrics that do not allow a direct comparison between OpenDaylight and ONOS (Khattak et al. 2014). Other references provide a direct comparison between both controllers but rely on older controller versions (Salman et al. 2016, for instance, present a comparison between OpenDaylight and Floodlight – the predecessor of ONOS). Due to substantial architectural changes in both controllers (MD-SAL change in OpenDaylight and the new distributed primitives in ONOS) such comparative studies may not be directly transposed to current versions. Furthermore, ONOS and ODL

have benchmark tests incorporated on their continuous development pipeline (*buildbots*) but not only test machines have different specifications but also the tests focus on aspects that do not allow to infer a comparison conclusion. Nevertheless, all those efforts have been contributing to standardize performance benchmarks and to the development of common evaluation tools. *CBench* (Trema 2018), initially developed for evaluating the Trema controller is now the *de-facto* tool to benchmark performance aspects of SDN controllers (Tootoonchian et al. 2012). The *CBench* test evaluates two performance metrics: throughput and flow set-up latency. In throughput mode, each *CBench* virtual switch constantly sends *Packet_in* messages to the controller and counts the number of corresponding Packet_out messages received from the controller. In latency mode, each *CBench* virtual switch sends a *Packet_in* to the controller and waits for the *Packet_out* message. The average flow setup latency is then calculated by *CBench* through evaluation of the delay between responses in the test period. Darianian 2017 performed the *CBench* experiment against current versions of OpenDaylight and ONOS. The testbed was composed of 3 Intel Xeon E5-2670 (24 cores @ 2.30 GHz, 256 GB RAM, 4 TB hard drive, and a 10 Gbps network connection for controller-to-switch communication achieved through a Cisco UCS 6248 switch). One server was used for *CBench*, while the other two were due to the installation of both controllers. Test results are presented in Figure 5.12. The authors also evaluated the effect of Hyper-Threading in controller performance.



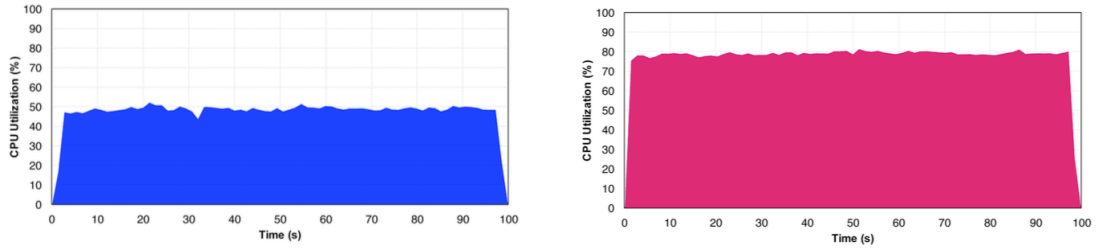
(a) Throughput comparison

(b) Flow setup latency results

Figure 5.12: *Cbench* test results for OpenDayLight and ONOS (plotted from data in (Darianian 2017)).

The authors' evaluation of ONOS and OpenDaylight through *CBench* indicates that ONOS outperforms OpenDaylight for throughput and latency tests. In fact, in throughput mode ONOS response scaled well with the increase of the number of switches. It reached a maximum plateau of 1.5 M responses/second in average. OpenDaylight, on the other hand showed performance degradation while the number of connected switches increased. Flow setup latency in ONOS was found to be nearly half of the latency of OpenDaylight. The presented results are interesting since both ONOS and ODL use the Netty framework for network I/O (UDP and TCP socket servers), so similar results would have been expected. We can infer the threading mechanisms of both controllers and the consistency models for their internal stores play a significant role on the overall controller performance. Hyper-threading brought improvements in both controller experiments but performance-wise differences remained the same. The authors also analysed the CPU usage of both controllers by grabbing

metrics from Netdata during test execution. CPU usage for both controllers with 8 switches are presented in Figure 5.13. The differences in *Packet_in* handling latency can also be seen on the CPU usage. It should be noted that ODL, unlike ONOS, serialize all the data to disk in order to enforce a strong consistency model. Hence, the obtained results were somehow expected.

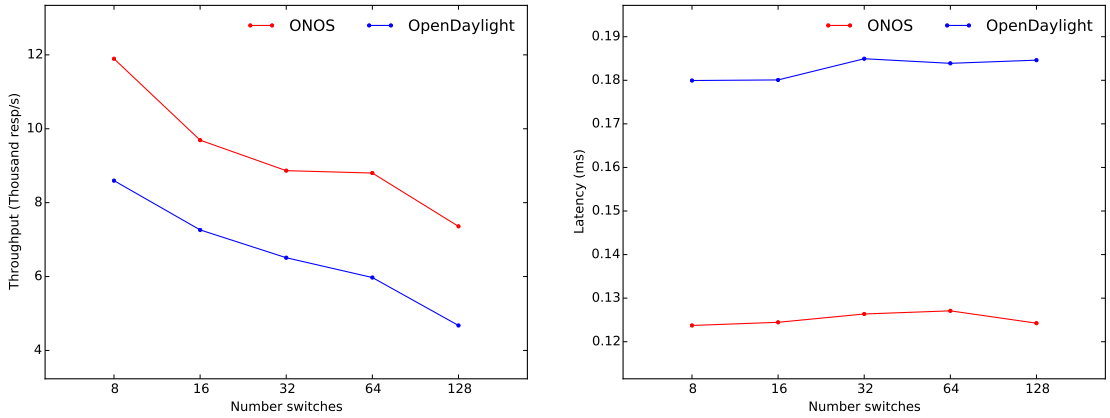


(a) ONOS CPU usage (b) OpenDayLight CPU usage

Figure 5.13: CPU usage of both controllers with 8 switches (from (Darianian 2017)).

Another important reference is *BTest*, an SDN controller benchmark tool proposed by the Politecnico di Milano (Cadenas et al. 2016). *BTest* is a *Cbench* extension to account for stress testing (perform latency and throughput tests over long periods of time) and automatic plotting supported through a web interface. Although not stating the specification of used machines, the test environment was kept consistent to perform tests on ODL (Helium version) and ONOS (version Falcon).

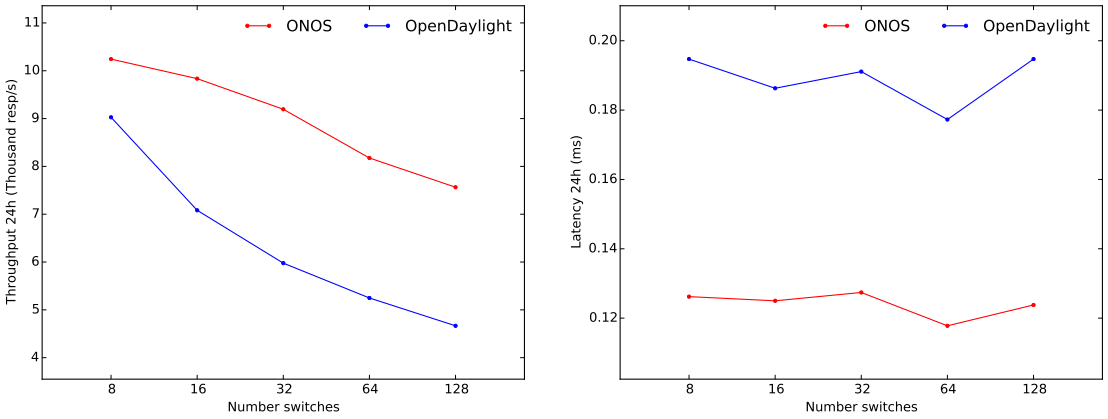
Throughput and latency test results are presented in Figure 5.14. The results of the same tests executed through a 24 hour interval (stress test) are presented in Figure 5.15.



(a) *Btest* - throughput mode (b) *Btest* - latency mode

Figure 5.14: *Btest* test results (plotted from (Cadenas et al. 2016)).

Obtained results are in-line with those of Darianian 2017: ONOS surpassed ODL in both latency and throughput tests. Note, however, that both tests only reflect the operation of a single controller node and completely neglect the clustering capabilities of both controllers. As the ONOS team states (ONOS-team 2016), new testing methodologies should be developed to better interpret the effects of clustering on SDN controller benchmarks. It is impossible to extrapolate performance of a multi-node controller from a single-node one, as



(a) Throughput mode 24 hours

(b) Latency mode 24 hours

Figure 5.15: *Btest* stress test results (plotted from (Cadenas et al. 2016)).

with multi-node controllers one can easily get worse performance or even poorer reliability. However, we can expect a ONOS cluster operating in the same conditions as a ODL cluster will show better performance metrics due to its internal eventually-consistent primitives.

5.1.4 Selected Network Controller

In Section 4.4.2 we state the SDN controller platform should be selected according to performance metrics. If we account for this and take benchmark results from Subsection 5.1.3 into consideration, ONOS is an obvious choice.

Moreover, in our opinion there are stronger facts which also lead us to ONOS. Unlike ODL, ONOS started from the early days as a distributed controller with clear quality attributes in mind: performance and availability. It is thus expected to be much more mature in what concerns its internal clustering mechanisms. Furthermore, those quality attributes are similar to the ones chosen for the development of the platform of this thesis. The architectural evolution of both controllers also shows different trends. The evolution of ONOS has been to improve its quality attributes while keeping a stable API. ODL, on the other hand, dramatically changed its API as an effort to keep a consistent development model where multiple partners can co-operate. By doing so, YANG (used by network operators with the NetConf protocol) was selected as the data model for the MVC design pattern, probably as a way to reduce the learning curve for network operators in the SDN transition. However, for someone without prior YANG or Netconf experience, the SDN application development experience can be cumbersome with OpenDaylight. ONOS, on the other hand, has a stable core which provide all the functionalities any network application requires.

ONOS also distinguishes itself from ODL on other important aspects in the context of this thesis. The distributed primitives give SDN developers more flexibility as a part of the application data can be stored in a non-consistent way to minimize the effects on performance. ONOS intents are also a key aspect for some design decisions in this thesis. *Host-to-Host* intents, for instance, automatically create host-pair flow rules leading for an easier way to create logical sub-networks than relying on VLAN tagging. Since sub-networks are constructed in the form of flow rules, an SDN-based IDS application can then change those

rules in a proactive way, to make network switches to provide multiple output ports per rule. The extensive existence of event types and the possibility of receiving those events in SDN applications by means of listeners also makes the development of an SDN probe (event factory) easier.
Considering all the advantages presented above, ONOS was selected as the SDN controller for the platform to be implemented within the scope of this thesis.

5.2 System Architecture overview

This section aims at providing a bird's-eye view over the architecture of the SDN subsystem. The design internals (lower-level architecture) of all the SDN applications are provided in section 5.4.

5.3 High-level architecture

Figure 5.16 presents the high-level architecture of the SDN subsystem of IADS. It is composed of 4 main components: a management web-interface, a distributed control plane, the field network and the virtualization infrastructure. The domain processor (see section 4.1) is responsible for receiving events produced within the SDN subsystem. However, its architecture definition is out of the scope of this thesis.

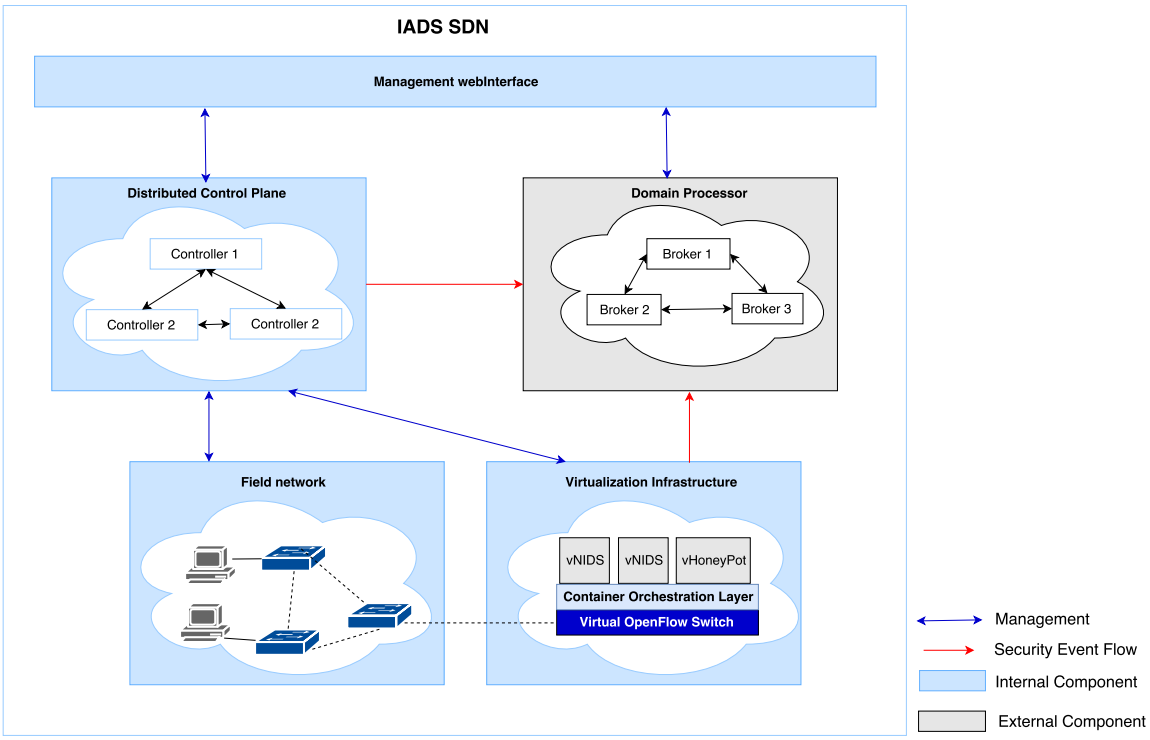


Figure 5.16: IADS SDN high level architecture.

The control plane is composed by a cluster of SDN controller nodes containing a set of specially crafted applications. Those applications enforce the goal of the control plane (managing the field network) through the installation of flow rules via the OpenFlow protocol. The virtualization infrastructure is a set of several physical computing nodes where virtual probes (virtual IDS, virtual honeypot) are deployed in the form of containers. Each virtualization host is composed by a containerization engine (Docker) responsible for orchestrating containers and a virtual switch (supporting the OpenFlow protocol) whose goal is to place containers "accessible" to the field network. Applications in the network controller issue container creation requests and, once they are accessible from the controller, program the network to provide copies of the network traffic to launched containers. In broad terms,

the distributed control plane can be seen as the SDN domain of the platform whereas the virtualization infrastructure represents the NFV domain. The management web-interface is responsible for providing a global administration interface to the whole platform (including the control plane, the underlying network, the virtual infrastructure and any other component in the IADS platform).

The architecture of each control plane node is presented in Figure 5.17.

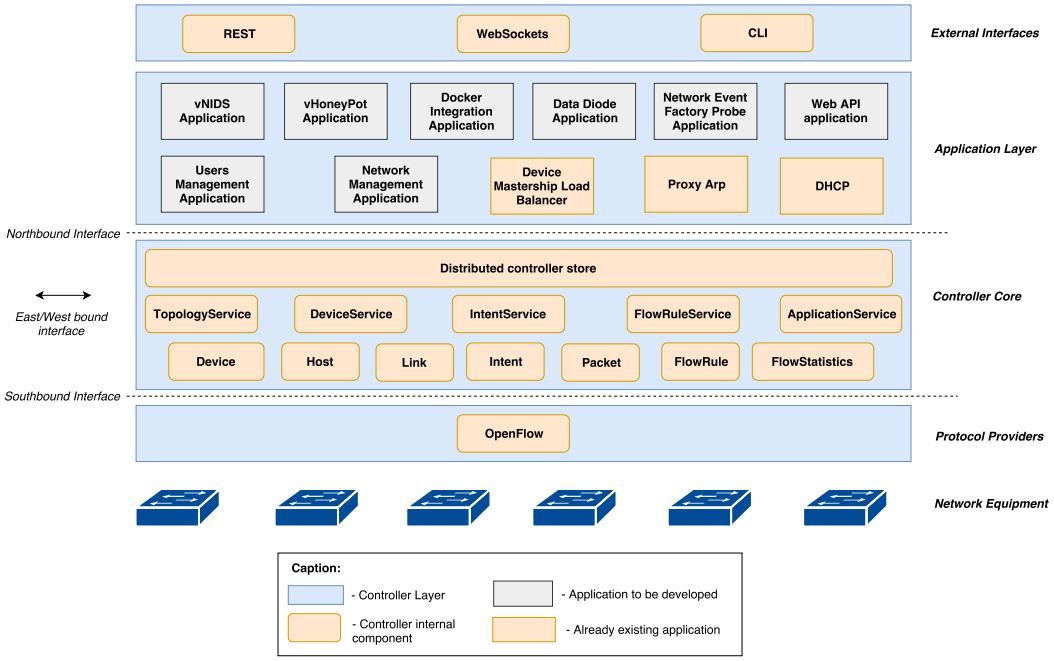


Figure 5.17: Distributed control plane node architecture.

As seen in Figure 5.17, the application layer of the distributed controller includes 11 applications. Three of them are already part of the ONOS controller and help to respond to some of the elicited requirements:

- **Device mastership load balancer** - This application automatically balances switch mastership across all available SDN controller nodes. It greatly improves performance, since management overhead is seamlessly distributed across all nodes. For instance, if we have 3 nodes and 3 switches, each switch will be configured by the controller to have a different master controller node. Redundant (and inactive) connections are still kept to other nodes in the controller cluster.
- **Proxy Arp** - An application that makes any ARP packet in the network to be sent to the controller. It is required so the global topology is constructed by the network controller, and to discover new hosts in the network.
- **DHCP** - A DHCP server implementation with SDN. Any DHCP requests are sent to the controller. The controller is also responsible for responding with an available IP address. This application is useful in the platform since it makes easy to attribute IP addresses to hosts in the network without the need of configuring each host with static IP information.

The remaining applications did not exist yet in the controller and were developed in the scope of this thesis - they are the core of the SDN subsystem of the IADS platform. Functional aspects of the platform (c.f. Section 4.2.7) map well to individual applications. By using multiple applications, quality aspects such as code modularity and interoperability are promoted within the platform. Below, a simple description of each application is provided. Note, however, that component and connector views are provided for each of these applications in Section 5.4, allowing a lower-level analysis of its architectural internals. Note also that in Figure 5.17, and contrarily to some references in the literature, all network logic is implemented in the application layer of the controller (using its core services) not relying on high-latency controller external interfaces (such as REST). This design decision greatly contributes for improving the performance of the subsystem.

- **Users management application** - This application is responsible for creating and storing user information (network admins, network tenants and security monitors) in the controller distributed store. It is also responsible for providing the core logic for user authentication.
- **Network management application** - Application responsible for creating logical sub-network in the overall network, by taking advantage of intent-based networking. Since the global topology graph is available at the controller, the application can install host-to-host intents as part of the definition of a sub-network. This application is also responsible for mapping each sub-network with a network tenant profile. Host, link, device information and statistics are also retrieved from the service this application exposes on the controller.
- **Docker integration application** - This application works as an API to access docker physical hosts from the network controller. Furthermore, it also implements an API to a private docker registry to upload docker template images. Other applications use the services exposed by this application to issue container start requests or to retrieve container network information as if they were regular SDN hosts. It is also through this application that physical docker nodes (and a registry) are associated with the SDN subsystem. Container statistics are also retrieved via this application. Any container launched from this application contains a label to indicate if it should be attached to the SDN network.
- **vNIDS application** - Application that implements the network logic for a virtual IDS service in SDN. This application issues IDS container start requests via the docker integration application (specifying which IDS image to use from the docker registry), modifies the rules installed by the network application so they have multiple output ports (a copy of the packet has to reach the container) and also installs lower priority rules to ensure any packet reaching any switch (and sent or received by the hosts being monitored) also reaches the launched container. Note an IDS in SDN is just a service definition. A network tenant can have multiple IDS services associated with a logical network, each one having different container images (e.g. one using Snort, other using Bro).
- **vHoneypot application** - Application which allows the user to specify an IP address range for the operation of a virtual honeypot. It requests a container start via the docker integration application and installs rules in the underlying switch infrastructure to forward any packets (whose destination is an IP in the specified range) to the launched honeypot container and vice-versa.

- **Data diode** - The data diode application is a simple SDN application which allows network tenants to block traffic in a specified edge link (the link between a host and device) in one of its directions. It works similarly to a physical data diode, however, it is a virtualized version implemented with SDN and OpenFlow.
- **Network event factory application** - This SDN application is a special probe in the IADS platform. It encodes any SDN event (device, host, link, topology, statistics) received in the controller and sends it asynchronously to the domain processor. Encoding is performed according to the ATENA data model (AtenaConsortium4.3 2017) using the Avro serialization format. The existence of this application allows the SIEM component of the IADS architecture (see section 4.1) to also employ machine learning algorithms in data originating on the control plane.
- **Web API application** - This application extends the REST and Websockets external interfaces of the controller in order to expose the services defined (and exported) by all the other applications. Having all the applications defining independent REST interfaces could have been a possible alternative. However, as the application layer of the controller scales, having all the REST definitions in one single place might reveal a better architectural decision on the long run. A consistent namespace can also be used to the whole platform. This application is accessed (via the REST and websockets interface) by the management web interface.

Network programming within the SDN subsystem is furthered explored in Chapter 7. Note that in Figure 5.17 OpenFlow is the only southbound protocol enabled in the controller. Additionally, all applications also expose their services to the controller command-line interface. The CLI interface enables quick testing for application logic and provides the network admin with an alternative to REST. Figure 5.18 illustrates the architecture of another important block in the IADS subsystem: the virtualization host architecture.

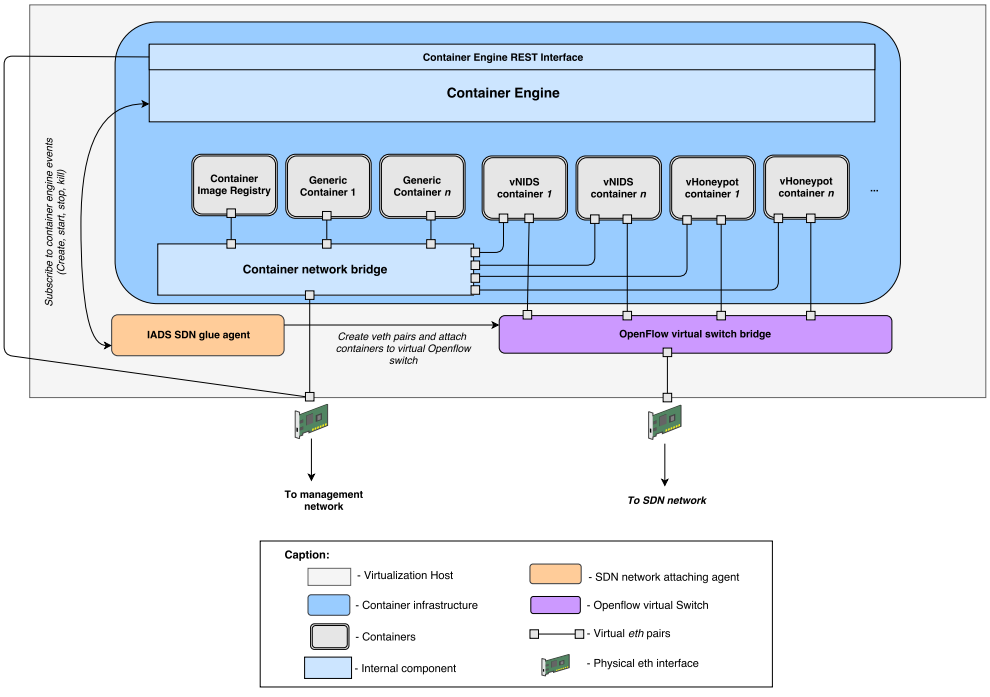


Figure 5.18: Virtualization host internal architecture.

Each virtualization node is required to have at least two Ethernet physical interfaces. One is connected directly to the SDN network while the other connects to the management network. The SDN interface enables virtual probe containers to receive network packets from the field network. The management network allows the network controller (more precisely the docker integration application) to request the start of new virtual probe containers. The virtualization node uses the Docker container engine to orchestrate the container lifecycle. This lifecycle is exposed to the SDN controller via its REST API implementation. The docker container engine also manages container networking: by default it creates a network bridge to which all containers are attached. This bridge makes it possible for containers to have access to the management network, meaning virtual probe containers can send events to the domain processor without changes to Docker. However, making virtual probe containers to receive network packets from the field network is not a trivial task, since Docker does not support OpenFlow by default. To accomplish this, each physical virtualization host has a virtual OpenFlow switch installed and a custom agent (SDN glue agent) developed specifically for this architecture. The SDN glue agent connects to the Docker API via its unix socket and listens for container-based events. Once a start event is detected and if the container contains a known label (e.g. `"-is-sdn=true"`) the agent creates a virtual ethernet interface, places it under the container Linux namespace and attaches it to the virtual switch bridge. The process is similar to the one described in Section 3.5.
The fact containers can belong to each of the two networks really entails one of the biggest advantages of the proposed architectures: the platform can use virtualization to efficiently manage its computational resources. Generic containers (e.g a database server) can co-exist with virtual probe containers within the same physical hardware. Recall from Section 3.5 container based virtualization has a lower resource footprint when compared to virtual machines. Hence, the whole platform is designed with performance in mind. To better understand the steps required for virtual probe deployment and how the several architectural blocks interact, a workflow diagram is shown in Figure 5.19.

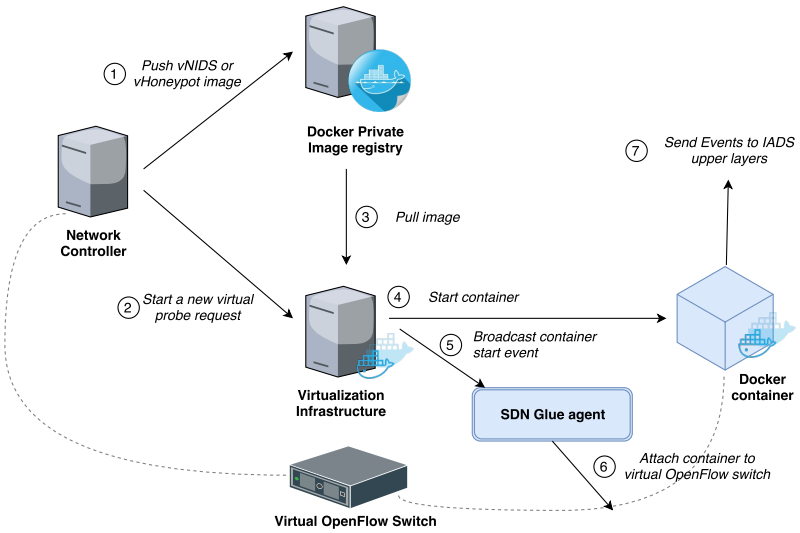


Figure 5.19: Probe deployment workflow.

The docker private registry in Figure 5.19 is a generic container which can run either on the virtualization infrastructure or in a special machine. It can also take advantage of the clustering mechanisms of Docker (e.g. Docker swarm) to be distributed. The private registry is based on the official Docker registry image template and stores all the container images

used for probes in different categories. For instance, the NIDS category groups templates specific for intrusion detection (e.g Snort) while the Honeypot category groups template images related to the vHoneypot (e.g. Honeyd). The registry can be seen as a "security probe" store.

To conclude the presentation of the high-level architecture of the SDN subsystem, an allocation architectural view is provided in Figure 5.20, identifying the used protocols.

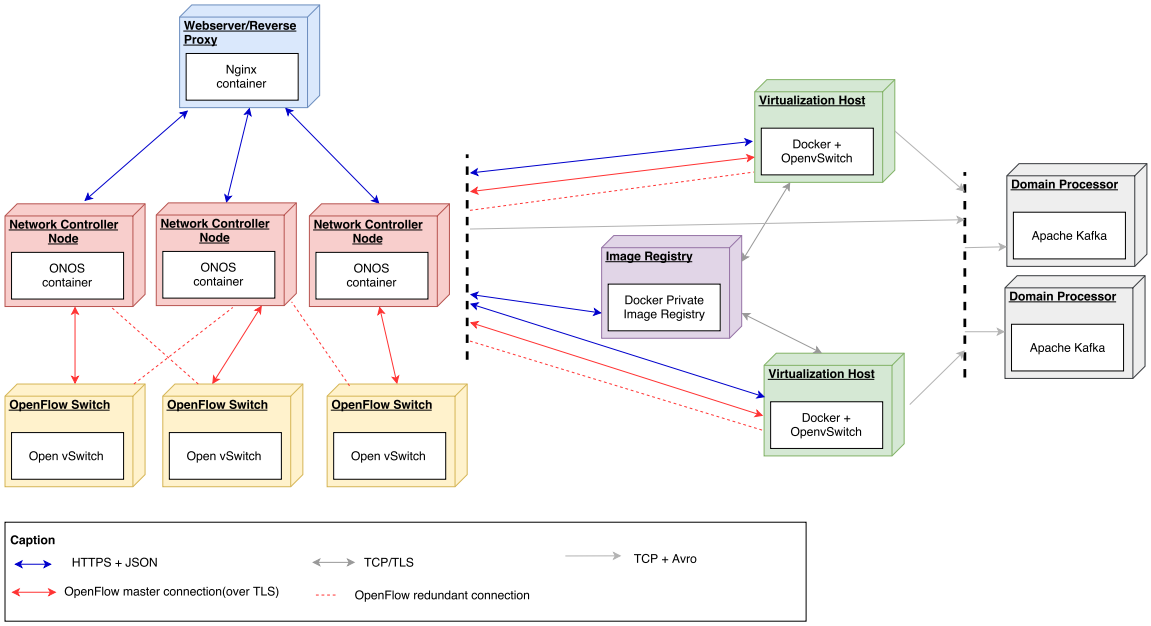


Figure 5.20: Allocation view.

The architecture goes a step further in the way it also sees internal components (such as the control plane) as micro-services. Each controller node is, in fact, a container running in general-purpose hardware. Deploying the control plane or migrating to more recent versions of the controller is easier due to containerization. It is also important to mention the web-interface is served by a Nginx container which also acts as a reverse proxy and load balancer. Note that the REST interface exposed from the network control plane (which the mentioned SDN applications extend) is available at any node. The NGINX container proxies REST API requests to the network controller using *round-robin* in order to distribute the load across all controller nodes. The architecture is also though to support encryption in every communication possible. Openflow switches can be configured to use TLS in the connection with the control plane. The docker registry is also configured with x509 certificates to communicate over HTTPS as well as the virtualization nodes.

5.4 System Applications and Components

For each application present in Figure 5.17 component and connector views are provided in this section, in order to deeply document the SDN subsystem architecture. The goal is to identify all the services each application exposes to the controller runtime as well as any dependencies on the controller core (accessed through the northbound interface).

Since SDN applications are OSGi bundles, a few notes are required before going into each application details. OSGi (OSGiAlliance 2018) has essentially two building blocks: services and components. An OSGi service is any object that is registered in the OSGi Service Registry and can be looked up using its interface name(s). An OSGi component tends to be an object whose lifecycle is managed, usually by a component framework such as Declarative Services (DS), Blueprint or iPOJO. OSGi components have some specificities:

- A component may be started and stopped.
- A component may publish itself as an OSGi service.
- A component may bind to or consume OSGi services.

In component-connector views provided in below sections, OSGi components and services are clearly identified. Other components referred just as "components" are internal functional blocks of the application (e.g.: the implementation of a client to an external system) required by published OSGi services and components. Each SDN application expose one or more services to the service registry. Those services can then be looked up in the OSGi service directory by other applications and their logic can be consumed by other applications. The services exported by each application are coloured in blue in the following diagrams. Note that, in the following figures, **ALL** the services and components above the northbound interface were developed by the author of this thesis.

5.4.1 Users Management SDN Application

The *Users management* SDN application expose an OSGi service (`UserManagementService`) which results from implementing the service definition interface. This is one of the simplest applications in the SDN-subsystem since it does not implement a distribured store. The application only exists as a means to validate users' provided authentication tokens and to provide the information contained within a valid token to the network controller. Authentication in IADS is executed by an external middleware component (IADS-auth) using JSON Web Tokens (JWT) - out of the scope of this thesis. The *Users management* application shares the same encryption key as IADS-auth to be able to validate provided tokens. The application makes use of ONOS `ConfigurationService` to let administrators configure the encryption key as an application property. ONOS `ApplicationService` is used to register the application in the controller.

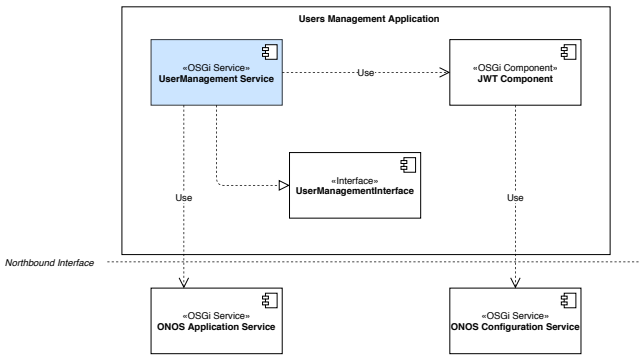


Figure 5.21: User Management application component and connector view.

5.4.2 Network Management SDN Application

Contrarily to the *Users Management* application, the *Network Management* application implements a distributed store. The store is used to store sets of hosts belonging to each subnetwork (and the network name). The store also maps network names to network tenants. To do this, the main `NetworkManagement` OSGi service depends on the `UsersManagement` service which is registered in the service directory by the *User management* application.

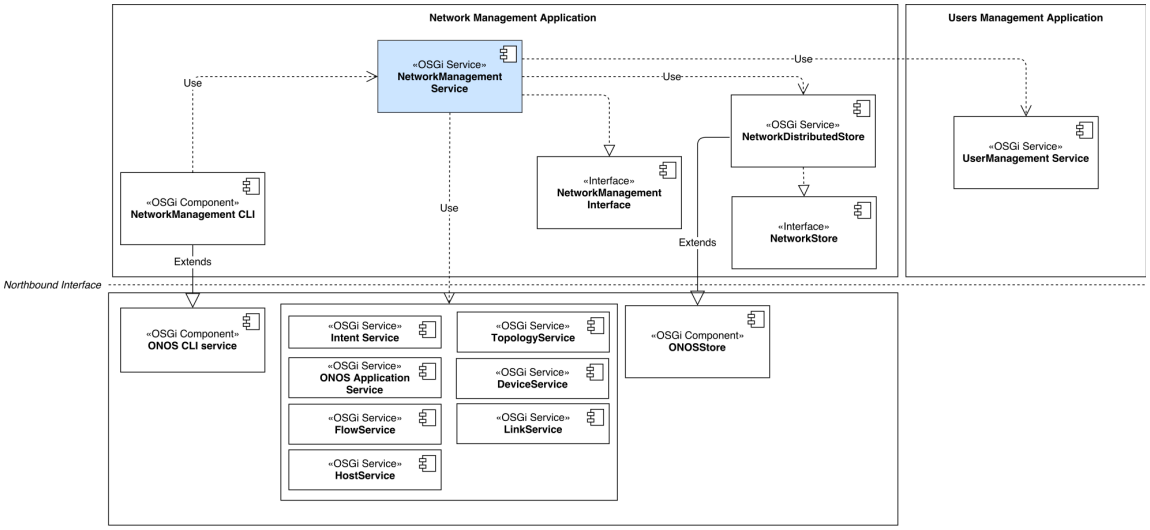


Figure 5.22: Network Management application component and connector view.

The application highly depends on ONOS core services. The `IntentService` is used so the `NetworkManagementService` can compute host-to-host intents based on the hosts belonging to each subnetwork. `TopologyService` is used for the application to compute the topology graph of the whole network or of logical sub-networks. The remaining services are used just as a means to proxy information from the controller (e.g. device, link and host statistics). Note that the network controller by itself cannot filter the network information of assets belonging to logical sub-networks (sub-networks are a concept of the IADS SDN subsystem). Hence, this application has all the logic required to reduce the network information available to users.

5.4.3 Docker Integration SDN Application

The *Docker Integration* application is just middleware that creates a common service to interact with the Docker container engine. Similarly to the *Users Management* application, it does not depend on other applications but it is itself a dependency for other applications. It exposes three OSGi services to the service register: `DockerClientService`, `DockerRegistryService` and `DockerNodeService`. The `DockerClientService` groups all the logic required to start, stop, pause, get logs or execute commands on containers. It uses the `DockerClient` component - an implementation of the Docker API via REST built from scratch. This service does not implement any store since it is simply an API to be used within ONOS.

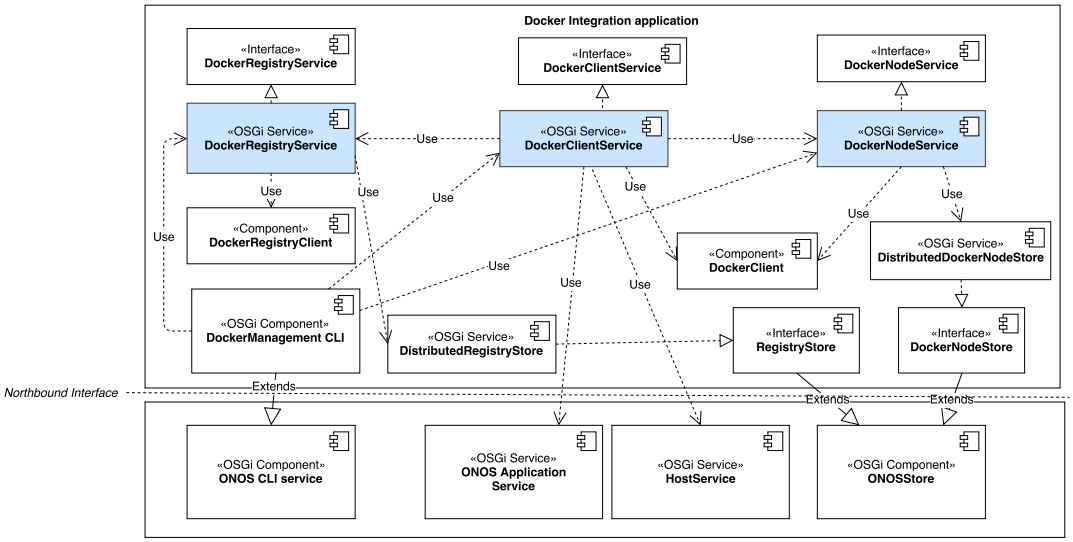


Figure 5.23: Docker integration application component and connector view.

The `DockerRegistryService` exposes the Docker registry REST API to ONOS. The communication between the registry and the controller is achieved with the `RegistryClient` component. The `DockerRegistryService` stores the information of the registry associated with the platform (IP address, port, username and password). The `DockerNodeService` is the service the application uses to associate or remove physical virtualization nodes from the platform. This information is stored in the `DistributedDockerNodeStore` OSGi service. Note that when an external application requests the start of a container it expects the Docker management application to respond with a `Host` ONOS object. In order to do this, the application relies on the `HostService` to find the Host object corresponding to the container (a lookup by IP or MAC address).

5.4.4 vNIDS SDN Application

The *vNIDS* application uses the services exposed by the *Users Management* application, *Network Management* application and the *Docker integration* application. It uses the `NetworkManagementService` and the `UserManagementService` to verify if the host requested to be monitored belongs to a tenant. Furthermore, it requests the topology graph of the sub-network also from the `NetworkManagementService`.

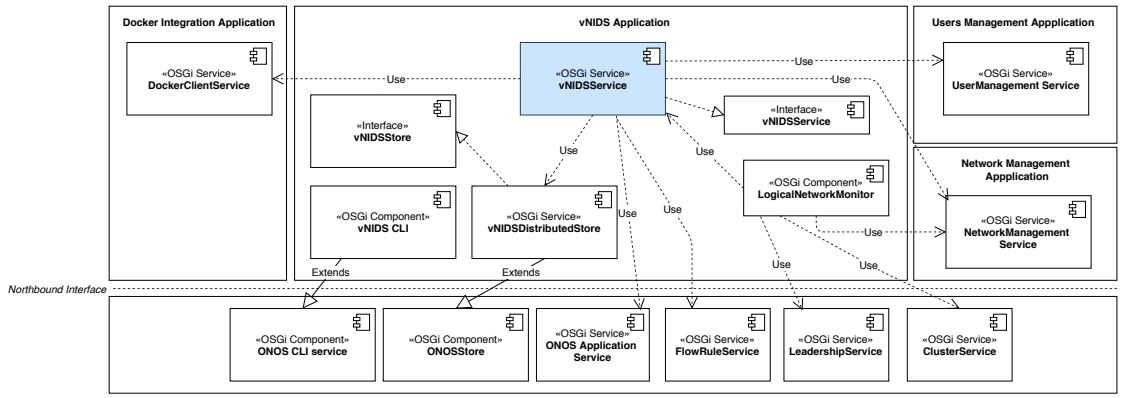


Figure 5.24: vNIDS application component and connector view.

It then requests the start of a new vNIDS container from the `DockerClientService` and receives the container information as an Host object abstraction. This abstraction allows the application to also find the edge device to which the container is connected. The ONOS `TopologyService` is used to find the path between the host being monitored and the container. For all of the devices in the path, the `vNIDSService` installs the necessary flow rules using ONOS `FlowRuleService`. Data (maps) for each service are stored in the `vNIDSDistributedStore` OSGi service. As logical subnetworks are subject to change (e.g a new host might be added to or removed from a given sub-network), the *vNIDS* application needs to be able to adjust dynamically to changes in the underlying network. The `LogicalNetworkMonitor` component is responsible for listening to `NetworkEvents` produced within the *Network Management* application. The component follows an observer design pattern, receiving any of the events (network created, removed and modified) asynchronously. Since there are multiple controller nodes in the cluster, the component makes use of ONOS *LeadershipService* to perform an election on a shared variable/topic to decide which node is responsible for performing the actions a given network event requires. `ClusterService` is used by the application to obtain the list of available controller nodes. In the case the leader fails, a new election takes place and another controller node will be responsible for mapping the application state with the state of the network. If a given host is removed from a network and a vNIDS service is deployed on the network, the event will be captured by the `LogicalMonitor`, the respective flow rules are removed for this particular host and the container may be destroyed. Using the same logic, when a network is destroyed, all the vNIDS containers associated with the network are also destroyed.

5.4.5 vHoneypot SDN Application

Architecturally, this application is similar to the *vNIDS* application. The main difference is that since the honeypot operations might require containers to have (or fake) IP addresses, the application must be able to lock them in the *DHCP* application so they won't get assigned to other hosts in the network. Flowrules are installed by the `FlowRuleService` and the topology graph is provided by the `TopologyService` to the `NetworkManagement` service of the *Network Management* application upon which this application depends.

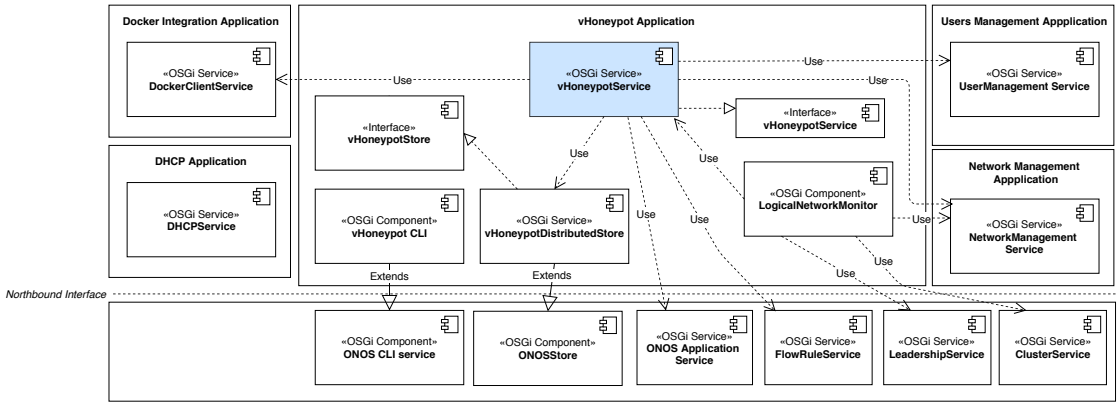


Figure 5.25: vHoneypot application component and connector view.

The application exposes a `vHoneypotService` and stores the information of each honeypot instance in a distributed store. It is important to note that the application also implements

a *LogicalNetworkMonitor* component that listens to `NetworkEvents` issued by the *Network Management* SDN application and adjusts the deployed services, depending on the event that occurred in the underlying network.

5.4.6 Data Diode SDN Application

The data diode application is the simplest application of all the three network security functions (vNIDS, vHoneypot and data diode). It just depends on the *FlowRuleService* to install rules in edge switches once a network tenant requests a link to operate as a data diode. It retrieves the Link object abstraction from the *NetworkManagementService*, programs the network and stores the instance data in a distributed store (`DataDiodeStore`).

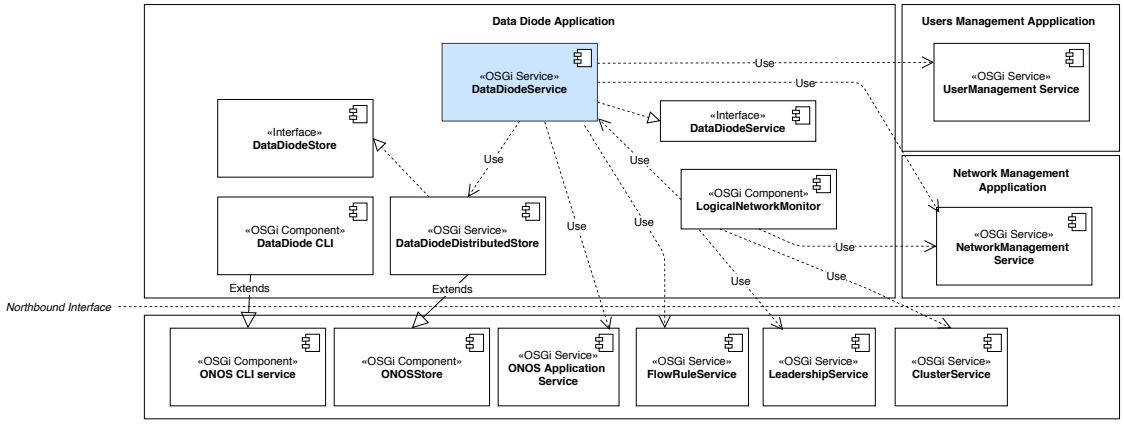


Figure 5.26: Data diode application component and connector view.

Similarly to the other two network functions, in order to react to network events and adjust the deployed services accordingly, the application also implements a `LogicalNetworkMonitor`. Note that commercial data diodes used in IACS provide several features that need additional support in software. The research paper presented in Annex D of Volume II discusses those features and details how they can be emulated in software.

5.4.7 Network Event Factory Application

The *Network Event Factory* probe application implements listeners/monitors for events published by the majority of ONOS OSGi services: `Topology`, `Device`, `Link`, `Host`, `Packet`, `FlowRule` and `intent`. Events are monitored by five OSGi components: `TopologyMonitor`, `DeviceMonitor`, `HostMonitor`, `LinkMonitor` and `ControllerMonitor`. Each of these monitors asynchronously receives events from the services enumerated above, encodes them in the Avro format (using the `IADSModelConverter` OSGi component) and dispatches messages to Kafka via the `KafkaPublisher` OSGi component. It is important to refer that due to the reasons mentioned before each monitor must also use the `ClusterService` and the `LeadershipService` to elect a leader for each of the event types. Hence, each monitor only processes the event in one of the controller nodes. The application configuration is not stored in a datastore but relies on the ONOS `ConfigurationService` directly. The application also defines a distributed store (`NetworkEventFactoryStore`) which is solely used to store counters for the events sent by the `KafkaPublisher` component.

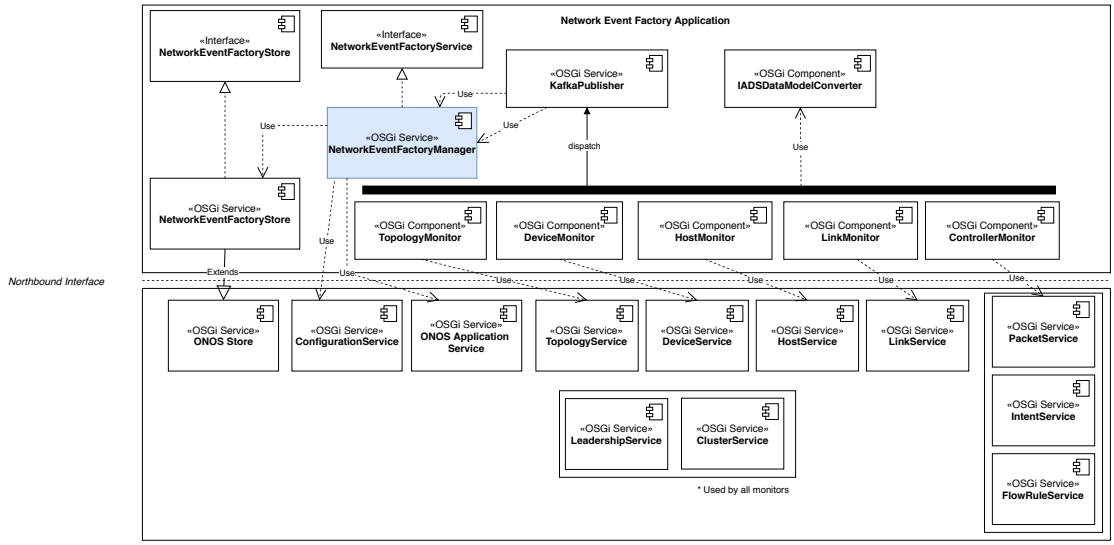


Figure 5.27: Network event factory component and connector view.

5.4.8 Web API Application

This application makes necessary methods and data from exposed OSGi services (those identified in blue in other component-and-connector views) accessible through REST and Websocket endpoints. The application contains a manifest that describes all the web servlets the application contains and their respective path. A simplified version of a component and connector view (assuming the exposure of one of the registered OSGi services) is presented in Figure 5.28. Authentication is achieved via JAX-RS annotations on each endpoint. Several filters (e.g. `@isAdmin` and `@isAuthenticated`) are available depending on the role of the user calling the endpoint. These filters make use of the `UsersManagement` OSGi service to validate the provided JWT.

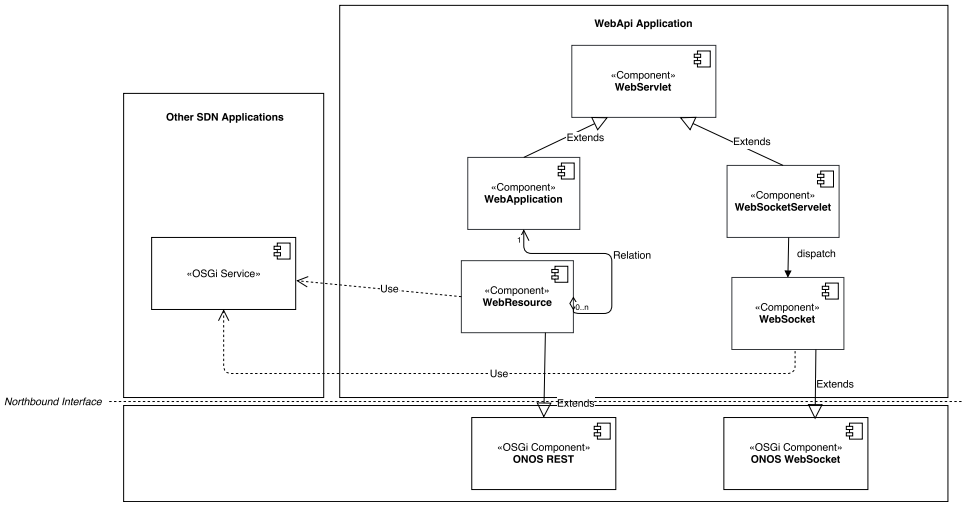


Figure 5.28: Web API application component-and-connector view.

The goal of this application is the creation of an abstraction layer that serves as middleware to the (web) User Interface and any other components that desire to consume services

belonging to the SDN network. It contributes to the centralization of all REST endpoints in a single location lowering the development and maintenance burden.

5.4.9 Management and Visualization Web-interface

The management web-interface is a single page application written in Javascript. Its internal architecture inherits from the used development framework: Vue-JS. Vue-JS empowers a model view-model view (MVVM) design pattern (Figure 5.29).

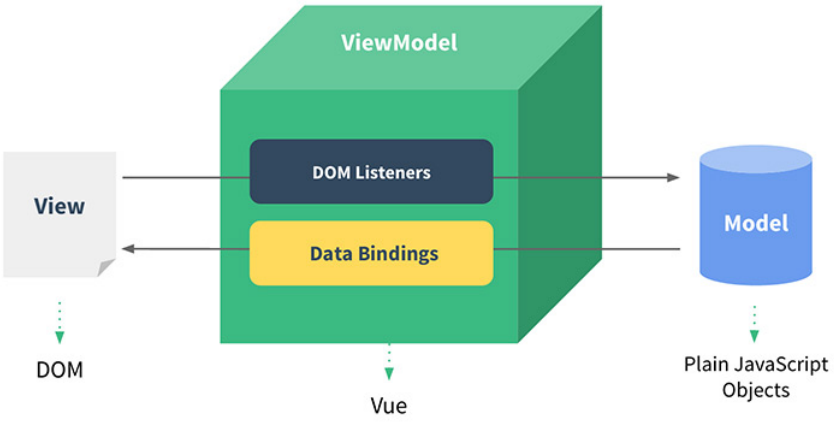


Figure 5.29: Vue-js MVVM design pattern, from Whatpixel.com 2016

The MVVM design pattern distinguishes itself from the MVC approach since it does not rely on a controller component to manipulate the Document Object Model (DOM). The Model contains data and some business logic while the View is responsible for the model representation. ViewModel (Vue) handles data binding, ensuring that the data changed in the Model is immediately affecting the View layer and vice versa. Thus, the Views in Vue-Js are data driven. Vue implements the pattern internally by setting reactive getters and setters for the elements in the View. Listeners (watchers) are set on the Model object. Once data changes in the Model, changes are notified to watchers which will trigger the reactive setter methods of DOM elements updating the view (Filipova 2016).

5.5 Chapter Wrap-up

This section was focused on the software architecture developed for the SDN context, extensively documenting its architecture, providing both a birds' eye view of its main components (control plane, virtualization infrastructure and management web interface) and a low-level architectural view of each developed application. It first starts by discussing the fundamental cluster mechanisms in distributed SDN controllers and the main differences between the most disseminated distributed controllers (ONOS and OpenDayLight) — while such content may at first seem to be more suitable for a state-of-the-art section, the reason for its inclusion has to do with the intention to detail the rationale and criteria used for the selection of the network operation system platform for the SDN subsystem. This selection process was highly dependent on the architecture of the controller and provided features, as

well as the implicit architectural trade-offs which strongly influenced the proposed architecture. The evaluation of existing network controllers allowed to conclude that ONOS, unlike OpenDayLight, has the same design goals as the quality attributes defined for the IADS platform. Furthermore, its documentation, performance and versatility are far superior than OpenDayLight. Thus, ONOS was selected as the base controller supporting the IADS SDN subsystem.

Chapter 6

Development Methodologies and Work Plan

This chapter is focused on the development and implementation methodologies and overall thesis work plan – the adopted development strategies. Section 6.1 details the applied software life-cycle and explains the implemented continuous-integration infrastructure which was later adopted by the IADS development team. Section 6.2 clearly identifies what was developed/implemented as part of this thesis and the reused software components. Section 6.3 presents the expected and actual work plan explaining the deviations. Development timelines per SDN application are also presented. Finally, Section 6.4 provides a final reflexion regarding planed (and achieved) goals clearly identifying which use-cases were not implemented.

6.1 Development Life-cycle

When developing large and complex software products, composed of several components, there is an implicit need to follow well-structured and defined software development life-cycles such as the waterfall model. In this kind of iterative lifecycles, a big part of the project's time is devoted to the requirements and design phases. This approach is expected to lead to the production of high quality software documentation artifacts and to a greater understanding of the software environment, but the development phase is deferred to later stages of the project timeline. It is common to also treat the development phase as a unique and monolithic stage. The usual waterfall model is shown in Figure 6.1.

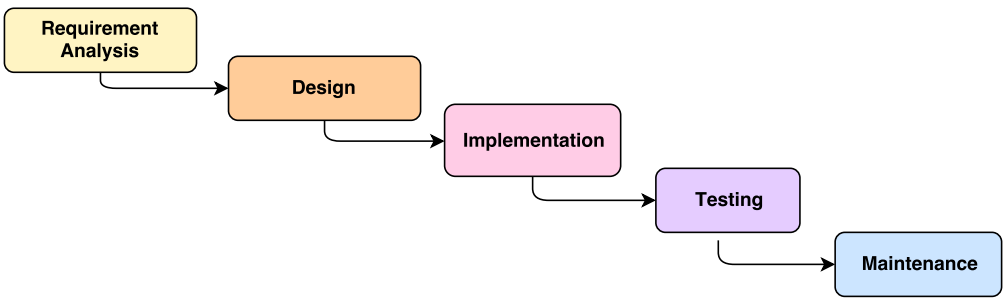


Figure 6.1: Waterfall software development life-cycle (Royce 1970) – (adapted from (Hughey 2017)).

Eric Raymond, in *"The cathedral and the bazaar"* (Raymond 2008), named this kind of approach to the development phases of a project as the cathedral-building style of development, since each release is built as a cathedral: the source code remains private to the developer and is released after all the implementation is done. By analyzing the overall quality of the software projects that adopted this kind of development style, the author concluded that this style of implementation did not reflect on the number of found bugs. Moreover, the author refers the amount of effort in the testing phase devoted to debugging between releases (usually under pressure due to time constrains) was sometimes impractical. Therefore, for the implementation strategy of this thesis, an alternative version of the waterfall method was followed – mixing the implementation and testing phases and breaking the system into its fundamental features. Similarly to waterfall, the same effort was given to the requirements analysis of the system (as provided in the form of use-cases) and to its architectural definition. However, the implementation and testing phases followed a *Release Early Release Often* (RERO) approach (Figure 6.2).

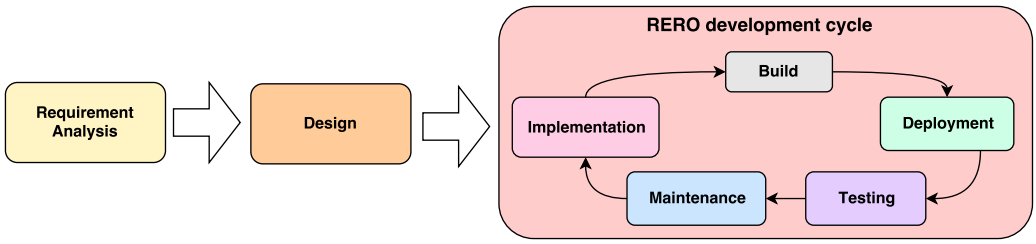


Figure 6.2: Modified waterfall lifecycle with a RERO approach in the development phase.

With the aid of continuous integration, and using component container virtualization, early versions of the software were released and deployed after each feature was implemented. This means that all the time between feature release was deferred to the execution of tests. The testing phase comprised some unit, integration, system and acceptance testing. The adoption of a RERO approach with continuous feature building and deploying helped automating some of the mentioned tests – mainly the execution of unit tests and other code quality approaches such as static code analysis. The project build is only successfully (i.e. results in a deployment) if the execution of unit tests passes. Unit tests for SDN applications test mainly the activation and deactivation of the application in the controller. Static code analysis consisted of using an external continuous code quality server (Sonarqube - (SonarSource 2018)) to find issues in the code base. Each commit to version control resulted in built application bundles (oar files) that were automatically installed in the production distributed control plane. Integration testing was greatly improved by the use of container based virtualization, since each component of the architecture was treated as a micro service. For example, a continuous integration job exists for deploying the control plane in a single click and install the current versions of all applications. Another CI job existed for removing all the probe containers from the virtual infrastructure so the test scenario can be restarted completely. For some components, the container deployment automatically means the component integration is verified. For instance, the web-interface needs to be "compiled" (uglify and minify the javascript code) and a new NGINX container needs to be generated. If the build passes, we can be sure that the integration of the code in the NGINX server was successful. System testing and acceptance testing relate normally to black-box testing approaches so, the early release of each feature allowed enough time to ensure the execution of tests on the deployed release until a new release was launched.

The term "Release Early, Release Often" was originally referred by Eric Raymond (Raymond 2008). Associated with a bazaar-like style of development, where each project member contributes code to the code-base, it often results in the release of versions untested / known beforehand to contain bugs. However, its early release gives beta-testers enough time to virtually find "every" bug.

Some of aforementioned advantages, along with other advantages provided by the RERO implementation approach are presented below (Haack 2011):

- Results in a better product.
- Results in a happier team by splitting the effort spent in the testing phase.
- Provides a rapid feedback loop so faster value is added.
- Gets feature and bug fixes in costumer hands faster.
- Reduces the pressure in the development team to make a release.
- Keeps the developer team always stimulated and rewarded.
- Makes the schedule more predictable and easier to scope.
- Lowers the barrier between the testing team and the development team allowing the first to follow the development efforts and to take a continuous part in the development process.

In this thesis we implemented all the continuous integration (and delivery) for the IADS platform. Specific build jobs existed for the control plane, the virtualization infrastructure, for the web interface as well as for other components of the IADS architecture. Figure 6.3 illustrate the overall automated process of continuous integration, testing and deployment we adopted.

Starting from the requirements elicitation and local development, all the code, typically small and verifiable changes, are committed to a Source Code management (a git server). Each commit will trigger a web hook in the Continuous Integration (CI) server, Jenkins instance responsible from static code quality auditing, testing and finally the deployment. To avoid excessive building times, dedicated building servers are used according the needs of each internal component. With the use of containers to ease the process of building, packaging and deployment, the build process consists in building a new container image for each component of the architecture. Then, those images can be pushed to different environments (testing, production) without the need of recompiling. An extra step of having a private and dedicated image registry was considered to make the move of images between servers, all the image builds are pushed to the Image registry, that stores specific image versions and tags. This private registry is only used for the CI infrastructure and should not be confused with the one used to store virtual probe images. Then, at the end of the building step, both testing and production environments pull the images from the registry. The development process loop is closed with the feedback observed in the production environment.

In alignment with the Open Web Application Security Project (OWASP) security guidelines (OWASP-Project 2017), the continuous integration process also encompasses an automated dependency check in its workflow (see Figure 6.4). This module is continuously updated in a daily basis, providing the means to seamlessly integrate the OWASP security compliance guidelines within the development process, avoiding common pitfalls, such as the use of dangerous dependencies or inadequate development practices. It is used to test dependencies

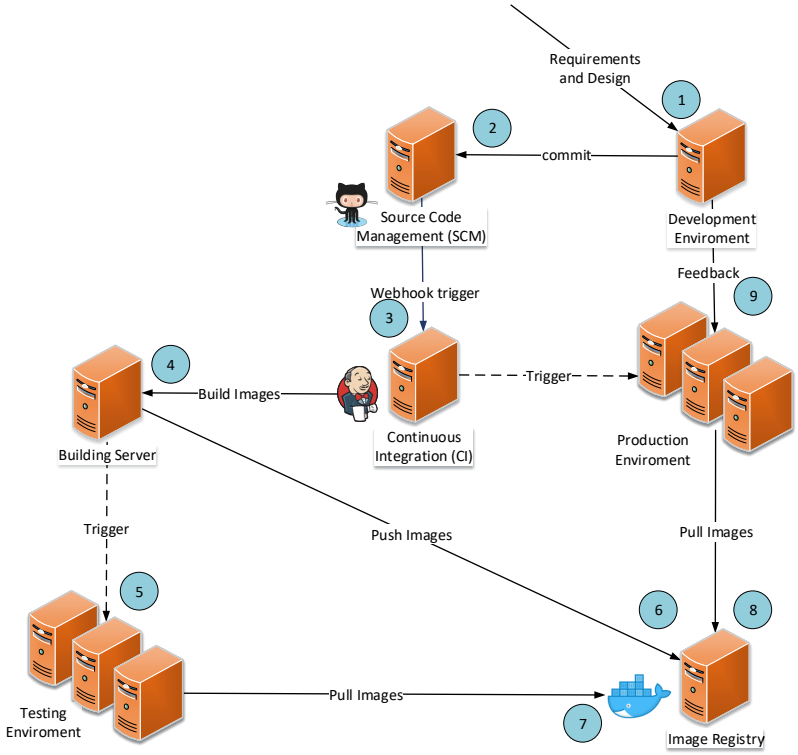


Figure 6.3: Continuous Integration, Testing and Deployment for the IADS.

of SDN applications (analysis of the maven XML manifest - pom.xml file of each application) and the website (package.json file) for known security vulnerabilities. The website deployed container (in production environment) is tested with the OWASP Zed proxy tool (OWASP-Zap 2018) to check for known web vulnerabilities (IP address disclosing, XSS, CSRF and SQL injection). This CI framework was firstly defined by the author of this thesis and later adopted by the whole ATENA team at the University of Coimbra.

Content of file package.json

Jquery is a javascript library for DOM traversal and manipulation, event handling, animation, and Ajax. When text/javascript responses are received from cross-origin ajax requests not containing the option `dataType`, the result is executed in `jQuery.globalEval` potentially allowing an attacker to execute arbitrary code on the origin.

```
{
  "name": "iads-webui",
  "version": "0.0.1",
  "description": "Atena IADS Web Application",
  "author": "DEI UC",
  "repository": "https://github.com/lmrosa/iads-webui",
  "license": "MIT",
  "scripts": {
    "dev": "node build/dev-server.js",
    "build": "node build/build.js",
    "unit": "cross-env BABEL_ENV=test karma start test/unit/karma.conf.js --single-run",
    "e2e": "node test/e2e/runner.js",
    "test": "npm run unit && npm run e2e",
    "deploy": "netlify deploy dist",
    "lint": "eslint -f checkstyle --ext .js,.vue src test/unit/specs test/e2e/specs -o eslint.xml || true"
  },
  "dependencies": {
    "axios": "^0.15.3",
    "babel-runtime": "^5.8.0",
    "chart.js": "2.3.0",
    "datatables.net": "^1.10.11",
    "datatables.net-bs": "^1.10.11",
    "faker": "^3.1.0",
    "hideseek": "^0.7.0",
```

Figure 6.4: OWASP dependency vulnerability analysis on the web-management interface.

6.2 Software artifact development and component reutilization

The design and implementation of the architecture and testbed of this thesis made use of several already existing software packages, namely:

- **OpenvSwitch** - Software-based switch implementation with OpenFlow support.
- **Open Network Operating System (ONOS)** - The distributed control plane of the SDN subsystem.
- **Docker** - Used for the containerization of virtual probes in the virtualization infrastructure and also to ease the deployment of the control plane.

In the control plane, ONOS, three already existing SDN applications (identified in Figure 5.17) were included in the architecture, to solve some of the project requirements, namely:

- **Proxy ARP** - To support the topology discovery within the controller.
- **DHCP** - To reserve static leases for virtual containers and to assign IP address for the topology hosts as an alternative to static IP assignment.
- **Mastership Load Balancer** - Used to automatically balance switch mastership across the cluster, improving the overall performance and distributiveness of the architecture.

Some other software components, on which the SDN subsystem depends or directly uses, were made by other team members for the ATENA project:

- **IADS-auth** - External authentication component of IADS. The SDN Users management application does not authenticate users but only validates provided authentication tokens previously attributed by the IADS-auth middleware.
- **IADS-avro** - A Java library for the encoding of events in the IADS datamodel (in Avro format), used in the *Network Event Factory* application.
- **IADS-management** - Middleware for probe and component configuration using the MQTT protocol that seamlessly integrates with the management web interface. Probes and components include a manifest file that defines which files (and regular expressions) are used for post-deployment variable configuration (out of the scope of this thesis).

For the intermediate review of the ATENA project, two of the testbed hosts were not developed by the author:

- **RapidScada Human-Machine Interface**.
- **Environmental Monitoring Unit (EMU)**.

All the other components of the architecture (and auxiliary tasks), listed below, were developed by the author in the ATENA project.

- **Control plane**:
  - Users management SDN application.
  - Network management SDN application.
  - Docker integration application.
  - vNIDS SDN application.

  - vHoneypot SDN application.
  - Data diode SDN application.
  - WebAPI SDN application (extending the external interfaces of the controller).
- **Virtualization infrastructure**:
  - SDN glue agent.
  - Probe containerization and adaption.
- **Visualization**:
  - SDN section of the management web-interface (circa 90%).
- **Automatic deployments**:
  - Continuous integration (CI) for SDN applications, SDN glue agent, web interface and components developed by others (e.g. IADS-auth).
  - Continuous code quality (Sonarqube) for the IADS-avro library and SDN applications.
  - Different pipelines for complete deployment and configuration of the SDN subsystem (and management interface) in Coimbra and IEC (Israel).

6.3 Work Plan

This thesis is affected by three main milestones:

- **M1** (on 14/11/2017) - A demo of the IADS platform was presented at the European Comission in the scope of the first project review. In the context of this thesis, a minimum viable product (deployment of a vNIDS through the web interface) had to be developed.
- **M2** (on 22/01/2018) - Intermediate thesis delivery
- **M3** (on 02/07/2018 or 03/09/2018 for the special season) - Final thesis delivery

Having started the work on the IADS platform early than the scheduling of the dissertation course, a total of 8 tasks were defined until the final thesis delivery:

- **Task 1** - Introduction to the ATENA project and the IADS platform. Familiarization with the state of the art (OpenFlow, SDN, NFV). Development of early stage prototypes to validate possible architectural options.
- **Task 2** - Requirements elicitation and formal definition of all IADS requirements for the project documentation (used in AtenaConsortium4.1 2017).
- **Task 3** - Architecture definition and test bed implementation.
- **Task 4** - First development and implementation stage:
  - *Users Management* SDN application.
  - Partial *Network Management* SDN application.
  - Partial *Docker Integration* SDN application.

  - *vNIDS* SDN application.
  - SDN glue agent.
  - Management web interface to support the above applications.
  - Concurrent implementation of the continuous integration infrastructure.
- **Task 5** - Writing of the first version of the thesis (intermediate thesis)
- **Task 6** - Second phase of development:
  - Refinement of the architecture and final testbed implementation.
  - *vHoneypot*, *data diode*, *network event factory*
  - Finish the *Network Management* application and the *Docker Integration* application with missing features to support the new functionality introduced by the new network services.
- **Task 7** - Testing and validation methodology development. Validation of the proposed architecture in terms of performance and availability. Preparation of the first round of research papers.
- **Task 8** - Writing of the final thesis.

The work timeline, proposed at the intermediate delivery of this thesis is represented in the gantt chart of Figure 6.5.

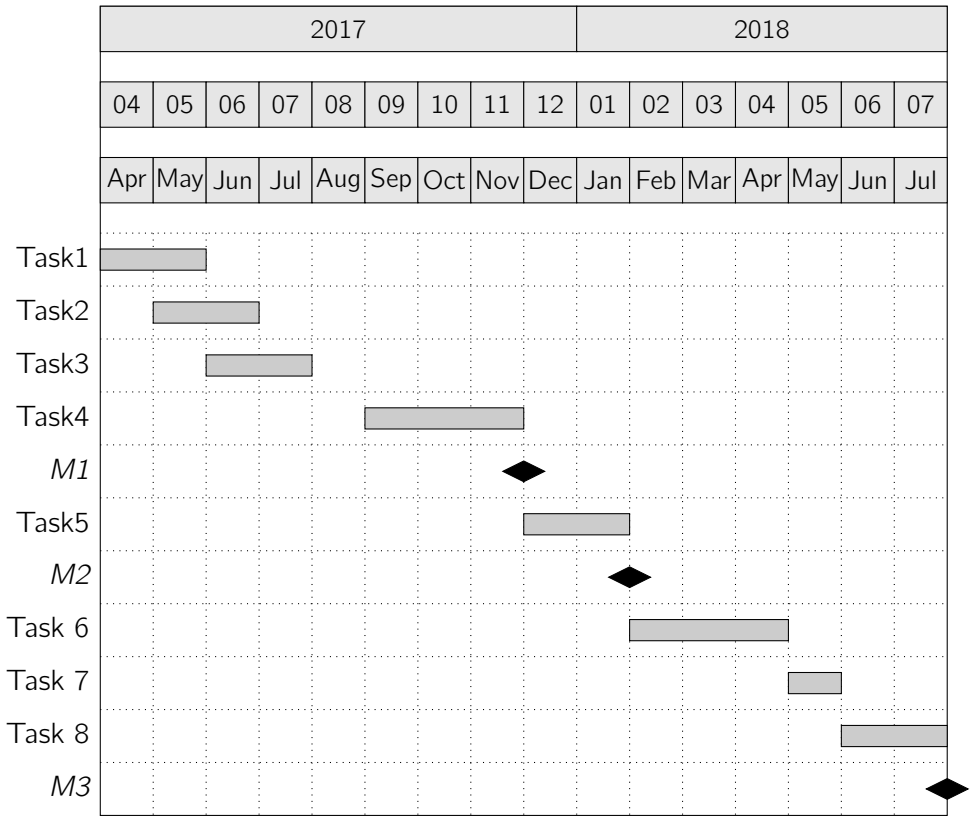


Figure 6.5: Gantt chart for the thesis activities (expected by the intermediate thesis delivery).

In reality, the timeline suffered some deviations, delegating the final delivery of the thesis to the special season (3/9/2018). The second phase of the development (and the conclusion of the applications partially developed on the first phase of development) took longer than initially expected. Note that building and deployment the applications (as well as application debugging) is a time consuming process, usually requiring complete cluster reboots. A new milestone (M4) was meanwhile introduced targeting the replication and extension of the developed architecture in one of the partners testbed (Israel Electric Corporation - IEC) for the final ATENA project review. The validation scenarios also took longer than expected due to the complexity of the overall time-based metric collection. Project deliverables and writing the companion research article also stole some time from the development and validation tasks. The final work plan of this thesis is presented in Figure 6.6.

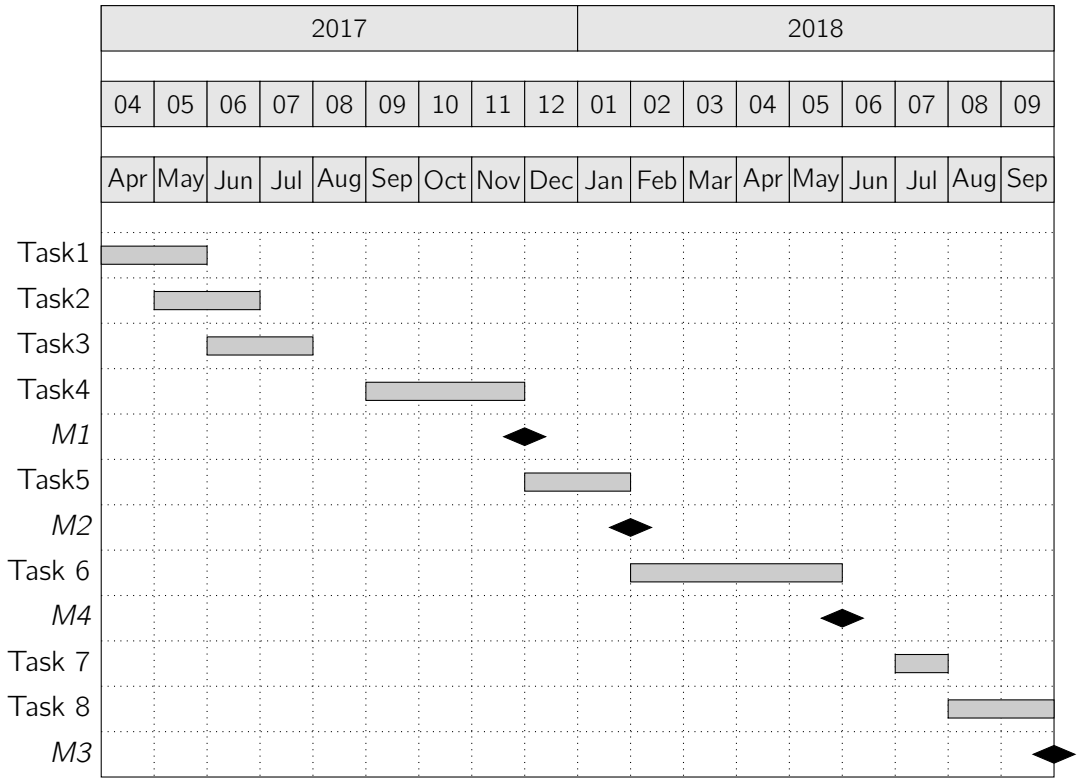


Figure 6.6: Gantt chart for the thesis activities (actual timeline).

6.3.1 Application development timeline

The gantt chart of Figure 6.7 specifies the effort spent per application and per development stage (Tasks 4 and 6). The chart was computed taking into account the overall *git commit* dates for each particular application/component. It is possible to see, unlike originally expected, that the docker integration application was the the one requiring more time to develop. In fact, despite the existence of two public Java libraries for the docker engine, one had to be built completely from scratch. Existing libraries depend on Jersey (which ONOS already uses) and could not be used in an OSGi environment, since it requires dependency injection at runtime. Existing libraries could not even be reused if bundled as dependencies (fat jar) in the docker integration application. As a result, a library was developed with the requirement of being as simple as possible in terms of dependencies, requiring only `java.net`.

The first stage of development did not include the ability of pulling images from the registry and required the images to exist in every virtualization node. In the second development stage, a client for the Docker Registry REST API also had to be built from scratch as well as the mechanisms for image pulling. The *WebAPI* application and the management web-interface development followed the development of all the other applications closely. Low level requirements (such as the information that should be made available from the WebAPI) suffered several changes throughout the project development. If virtual security services are compared (vNIDS, vHoneypot and data diode), it is possible to note all of them took more or less the same time to be built. vNIDS took more time since it was the first application to be developed, while the others required changes in the *Network Management* application or *SDN glue agent*, leading to a similar development effort.

6.4 Final reflections

Considering the 139 use-cases proposed for the SDN subsystem (c.f. Section 4), the development effort can be considered successful since only 17 use cases were not implemented. Use cases for user registration (*Users Management* application) were initially developed and a part of the application but were removed from the final version and implemented in a common IADS component (IADS-auth). Table 6.1 lists the use cases that were not implemented along with the specific reasoning.

Table 6.1: Use cases not implemented in the final delivery.

Use Case	Reasoning
NM_FR6	Renaming a created logical network was a low priority (CH) requirement. Logical networks are supposed to have a small number of hosts while the functionality is redundant if we consider the network can be removed (and a new one created with a different name)
NM_FR14-21	Use cases relate to features already available in the controller web interface. Functionality is redundant since the management web interface already provides the topology graph with filtering capabilities
VN_FR8	This use case is out of the scope of this thesis. Probe configuration was implemented in IADS-management in another internship.
VN_FR9-11	To properly implement scalability policies for the vNIDS service more development time would be needed. Between scaling the virtual service and create new ones (vHoneypot, data diode, network event factory) the author opted for the last so more value can be added to the final ATENA review.
NS_FR2-4	Statistics are already available in the controller web interface. The network event factory pipes all the statistics to the upper layers of IADS where further processing is required to take advantage of this data for any meaningful alarm. Having the host statistics in the web interface would not add any value to the user. Probe containers already show statistics for monitored hosts.
IADS_NFR18	Use case relates to scalability policies

Globally, the author considers the project timeline was adequate (even considering the deviations), if the required work load is taken into account.

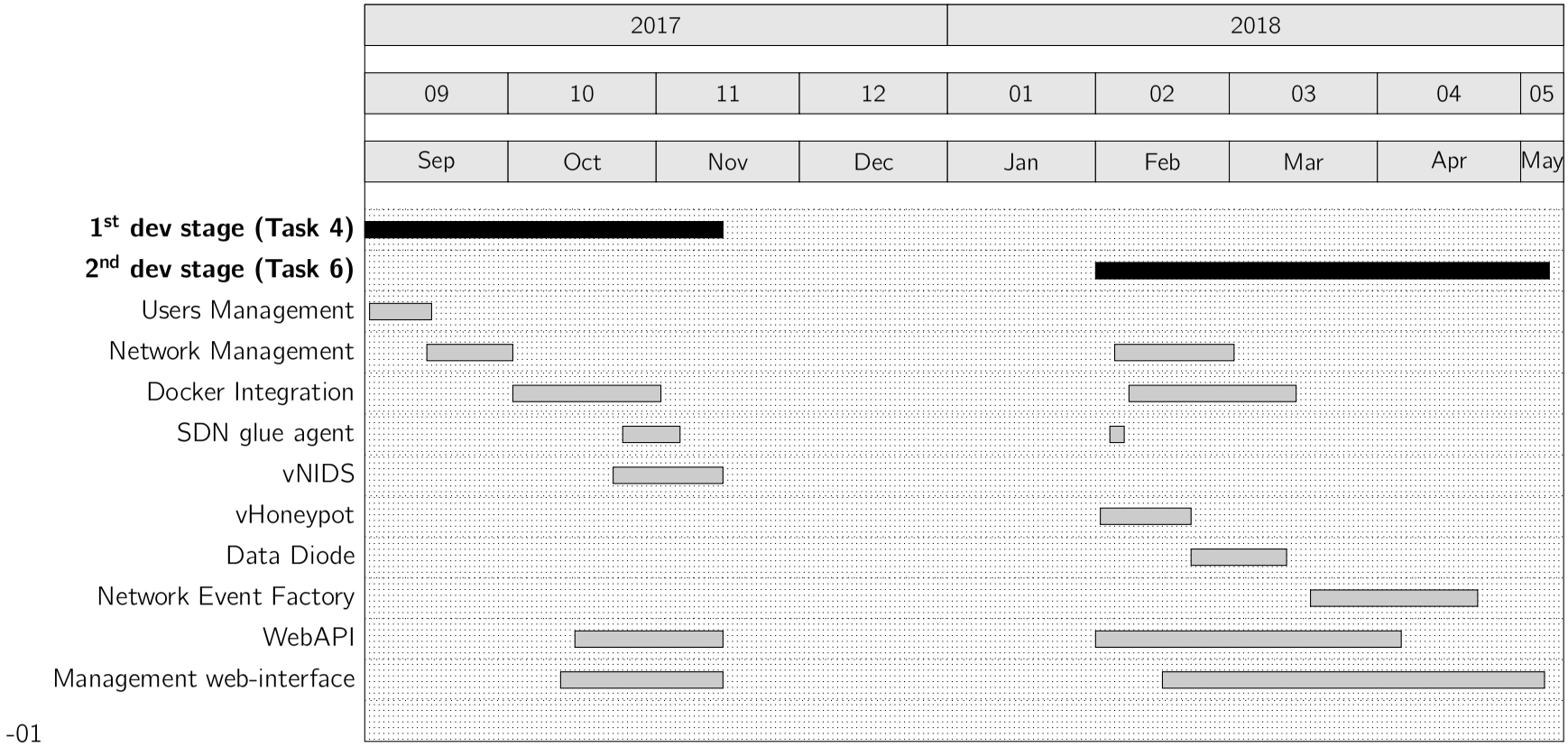


Figure 6.7: Development effort per application.

Chapter 7

Development and implementation Notes

Apart from the research component, this project has a strong development and implementation component. Thus, this chapter discusses the development decisions and the overall development path taken in each particular section of the architecture. Section 7.1 explains what was done on the data plane side to increase the performance of the subsystem. Section 7.2 explains network programming from the point-of-view of the developed SDN applications as well as the chosen algorithms. Section 7.3 focus on the design of the datastores for the SDN applications and how they relate, despite being self-contained in each individual application. Subsection 7.4 details the developed external interfaces, their use and consumption. Section 7.5 explains the work performed on the virtualization infrastructure: the development of the SDN glue agent and the work done on porting probe images to the platform. Section 7.6 targets the management web-interface development. Finally, section 7.7 provides a wrap-up of the chapter.

7.1 Data plane

The work performed on the data plane was mostly the installation, configuration and reutilization of already existing software packages. The OpenvSwitch project is a production-level, multilayer open-source virtual switch implementation. It is the de-facto software switch for OpenFlow, being extensively used in research projects. It is also the default virtual switch in the Citrix XEN hypervisor. OpenvSwitch was used for the testbed implementation, either in a bare-metal setup (a server operating as a switch) or virtual (in the virtualization infrastructure as another layer to manage container networking). It was configured with support for the *Data Plane Development Kit* (DPDK) for faster packet processing (TLF 2018). DPDK is a direct memory access (DMA) framework that enables the OSI application layer to directly access the buffers of the network interface card, bypassing the Linux kernel and avoiding expensive memory copy operations. DPDK is backed by the Virtual Function I/O (VFIO) universal driver and requires the setup of hugepages. Setting up OpenvSwitch with DPDK support was not a trivial task as the management of the network interface cards is no-longer achieved by (nor visible to) the Linux kernel.

7.2 Control plane network programming

Section 5.4 shows the low-level architecture of each developed application, identifying its component and OSGi service that are used from the controller core layer. However, it lacks a detailed overview over the network programming process and how it translates into flow-rules at the switch level. Hence, this section details network programming in the IADS SDN subsystem and briefly describes the chosen algorithms. Each developed SDN application, depending on its purpose, applies a different set of OpenFlow flow rules to the underlying switch fabric. Note that the *Docker integration* and *Network Event Factory* applications are middleware applications that interface either with Docker (virtualization infrastructure) or Kafka (domain processor). As a result, they do not program the network and are not referred in this subsection. To better explain what happens in the background, the topology presented in Figure 7.1, containing two hosts and an already deployed vProbe container (in the virtualization infrastructure), is used for exemplification purposes. The figure also presents each connection point (port to which any host is connected to each switch). Note that despite the topology simplification, the SDN subsystem is prepared to work on top of any topology, since connection points are obtained *"on-the-fly"* from the topology graph provided by the controller core layer.

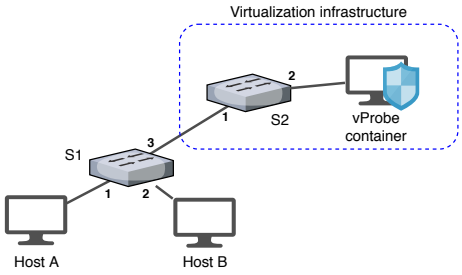


Figure 7.1: Example topology with two hosts and a vProbe.

Table 7.1: Host information.

Host	MAC address	Connection point
Host A	00:0C:29:26:E4:49	S1-1
Host B	00:0C:29:2B:99:F7	S1-2
vProbe	00:00:00:00:00:0A	S2-2

When a switch is first associated with the control plane, the ONOS controller, via the *Proxy ARP* application, automatically installs three rules on the switch to forward any ARP, LLDP (Link Layer Discovery Protocol) and BDDP (Broadcast Domain Discovery Protocol) network packets to the controller. Thus, the flow table of each switch is similar to Table 7.3.

Table 7.2: Default switch flow table.

Priority	Match Fields	Action	App name
40000	ETH_TYPE:arp	OUTPUT:CONTROLLER	core
40000	ETH_TYPE:lldp	OUTPUT:CONTROLLER	core
40000	ETH_TYPE:bbdp	OUTPUT:CONTROLLER	core

LLDP is used to discover direct links between switches and BDDP is used to discover the switches in the same broadcast domain. ONOS periodically (every 5 seconds) injects LLDP and BBDP packets containing a unique ID through all switch output ports. When packets are sent back from the switch to the controller, it becomes possible to know the packet path and determine the infrastructure topology. Similarly, before any IP communication occur between

hosts, ARP packets have to be generated by the communication hosts. ONOS instructs the switch to send ARP packets to the controller and the *Proxy ARP* application floods the packet through all the switch ports. This allows the controller to identify all network hosts (including vProbes) and correctly identify their location within the topology graph. The following subsections detail how flow-rules are installed per developed application, i.e. per deployed network service.

Network Management Application

The *Network Management* application relies on ONOS intent framework to create logical topologies. Considering the example topology of Figure 7.1, the application asks for the creation of a *HOST-TO-HOST* intent between Host 1 and Host 2 when a request for a logical network creation is placed. ONOS internally compiles the intent, finds the connection points and the path between hosts and translates it into flow rules installed on all applicable switches. In this case, since both hosts are attached to the same switch (S1), flow rules are only installed in this switch with the following fields:

Table 7.3: Network Management application intent translation.

Priority	Match Fields	Action	App name
...	...	...	core
100	IN_PORT:1, ETH_DST:00:0C:29:26:E4:49, ETH_SRC:00:0C:29:2B:99:F7	OUTPUT:2	intent
100	IN_PORT:2, ETH_SRC:00:0C:29:26:E4:49, ETH_DST:00:0C:29:2B:99:F7	OUTPUT:1	intent

It is important to note that the *Intent framework* was selected instead of individual rule installation since ONOS automatically tracks any installed intent state and readjusts them in case of a switch or link failure.

vNIDS Application

In OpenFlow, if a packet reaching a switch flow table has match fields aligned with more than one flow rule, only the rule with higher priority is triggered. As a result, the *vNIDS* application could not simply install another rule with a different port as connectivity between hosts would be disrupted. Two alternative approaches could have been used:

(a) Install a rule similar to the intent, with a higher priority.

(b) Modify the treatment/action of the pre-installed rule.

We decided to use (b), since in our opinion network connectivity should take precedence over packet monitoring (availability is the fundamental requirement of IACS). Also, with (a) we would increase the amount of installed rules making one of them redundant. Intent readjustment would also be lost if (a) was used resulting in service disruption in a switch/link failure case.

Additionally to changing the previously installed intent flow rules, the application also installs other flow rules (with lower priority) to guarantee that any packets generated or received by the hosts being monitored are also redirected to the vProbe container. Note that when monitoring host traffic, although connectivity is restricted by the logical network definition (host-to-host intent pairs) and packets which header does not contain the match fields for base connectivity cannot reach any host, there is an interest in monitoring those packets from a cybersecurity standpoint. Hence, for switch S1, the flow table after adding both hosts to a vNIDS service is shown in Table 7.4.

Table 7.4: vNIDS installed flow rules for switch S1.

Priority	**Match Fields**	**Action**	**App name**
...	...	...	core
100	IN_PORT:1, ETH_DST:00:0C:29:26:E4:49, ETH_SRC:00:0C:29:2B:99:F7	OUTPUT:2,3	intent
100	IN_PORT:2, ETH_SRC:00:0C:29:26:E4:49, ETH_DST:00:0C:29:2B:99:F7	OUTPUT:1,3	intent
80	ETH_DST:00:0C:29:2B:99:F7	OUTPUT:3	vnids
80	ETH_DST:00:0C:29:26:E4:49	OUTPUT:3	vnids
80	IN_PORT:1, ETH_SRC:00:0C:29:2B:99:F7	OUTPUT:3	vnids
80	IN_PORT:2, ETH_SRC:00:0C:29:26:E4:49	OUTPUT:3	vnids

Similarly, in switch S2, the same low priority rules are installed as shown on Table 7.5. Note in this case there are no rules installed by the *Network Management* application as Host1-Host2 base connectivity does not traverse this switch.

Table 7.5: vNIDS installed flow rules for switch S1.

Priority	**Match Fields**	**Action**	**App name**
...	...	...	core
80	ETH_DST:00:0C:29:2B:99:F7	OUTPUT:2	vnids
80	ETH_DST:00:0C:29:26:E4:49	OUTPUT:2	vnids
80	IN_PORT:1, ETH_SRC:00:0C:29:2B:99:F7	OUTPUT:2	vnids
80	IN_PORT:1, ETH_SRC:00:0C:29:26:E4:49	OUTPUT:2	vnids

The developed algorithm is presented below in simplified pseudo-code. The application performs additional checks to ensure the same traffic is not replicated multiple times in the path between the host and the vProbe container.

Algorithm 7.1: vNIDS algorithm exemplification.

```
1  input: hostId, vProbeId, hostMAC, topologygraph
2
3  begin
4  hostSwitch, hostPort = get_connection_point(hostId)
```

```
5   vProbeSwitch, vProbeport = get_connection_point(vProbeId)
6
7   for switch : topologygraph
8     do
9       rules = find_intent_rules(hostMAC)
10      if rules
11        port = get_connection_points(switch, vProbeSwitch)
12        for rule : rules
13          do
14            modify_rule_with_new_output_port(port)
15          done
16      install_low_priority_flow_rules()
17    done
18  end
```

vHoneypot Application

When a vHoneypot container is deployed, the resulting host does not have any connectivity with the remaining network hosts. Hence, the goal of the *vHoneypot* application is to ensure every host is able to reach the container. Table 7.6 shows the installed flow rules in switch S1, considering the vHoneypot container has a MAC address of `00:00:00:00:00:0A` and a fake IP address of `192.168.1.137`. In generic terms, for each host in the logical network, the application finds the switch to which the host is connected and computes a path leading to the running container. For each switch in the said path, rules are installed to forward traffic from the input port to the output port (and vice-versa), considering the MAC addresses of the host and the container. Since vHoneypot containers can fake IP addresses, rules are also installed to forward traffic to any fake IP addresses the container is configured to use. The following pseudocode illustrates the process for a single host.

Algorithm 7.2: vHoneypot algorithm exemplification.

```
1   input: host, vProbe, fakeips, topologygraph
2
3   begin
4   hostSwitch, hostPort = get_connection_point(host.Id)
5   vProbeSwitch, vProbeport = get_connection_point(vProbe.Id)
6
7   path = get_path(hostSwitch, vProbeSwitch) in topologygraph
8   for switch, connection_point in path
9   do
10    forward_traffic(host.MAC, vProbe.MAC, connection_point)
11    for ip : fakeips
12    do
13      forward_traffic(host.MAC, ip, connection_point)
14    done
15  done
16  end
```

For switch S2, rules are the same as presented in Table 7.6, although with port 1 and 2 both replaced by port 1, and port 3 replaced by port 2.

Table 7.6: vHoneypot installed flow rules for switch S1.

Priority	Match Fields	Action	App name
85	IN_PORT:1, ETH_DST:00:00:00:00:00:0A, ETH_SRC:00:0C:29:2B:99:F7	OUTPUT:3	vhoneypot

85	IN_PORT:2, ETH_SRC:00:0C:29:26:E4:49, ETH_DST:00:00:00:00:00:0A	OUTPUT:3	vhoneypot
85	IN_PORT:3, ETH_DST:00:0C:29:26:E4:49, ETH_SRC:00:00:00:00:00:0A	OUTPUT:2	vhoneypot
85	IN_PORT:3, ETH_DST:00:0C:29:2B:99:F7, ETH_SRC:00:00:00:00:00:0A	OUTPUT:1	vhoneypot
85	IN_PORT:1, ETH_SRC:00:0C:29:26:E4:49, ETH_TYPE:ipv4 IPV4_DST:192.168.1.137/32	OUTPUT:3	vhoneypot
85	IN_PORT:2, ETH_SRC:00:0C:29:26:E4:49, ETH_TYPE:ipv4 IPV4_DST:192.168.1.137/32	OUTPUT:3	vhoneypot

Data diode application

Since the goal of the *Data diode* application is to block traffic, we chose a priority (200) higher than the one provided by the *Network Management* application (100) for any application installed rules. As multitenancy is a requirement of the SDN subsystem, rules cannot be simply installed to drop all traffic at a port if a data diode is instantiated. An host is an asset that can belong to multiple networks while data diodes are virtual services which only apply to a single network. Dropping traffic at an edge link would disrupt connectivity in other logical networks. Hence, the network programming of the *Data diode* application is the following:

- Find the edge switch of the host which is required to have an uni-directional link.
- Install rules to drop traffic depending on the direction of the unidirectional link taking into account the MAC addresses and input/output ports of all hosts in the sub-network.

For the topology of Figure 7.1, and considering a data diode is deployed in the *Host A - S1* edge link (making Host A a transfer only - TX - device), the resulting flow table of S1 is as simple as the one presented in Table 7.7.

Table 7.7: Data diode application flow rule programming.

Priority	Match Fields	Action	App name
200	IN_PORT:1, ETH_DST:00:0C:29:26:E4:49, ETH_SRC:00:0C:29:2B:99:F7	DROP	data diode
100	...	...	intent

By selecting an higher priority for data diode flow rules, when a packet that matches both MAC addresses reaches the S1 switch it will automatically be dropped regardless of the rule

installed by the *Network management* application (intents). It is also important to note that, by default, if the packet matches no rule, the switch simply drops it as well.

7.3 Application datastores

ONOS makes use of MapDB for its internal distributed datastores. MapDB is a simple NO-SQL database that transforms standard Java data structures (maps and lists) into document models using different distributed primitives. Applications may decide to store the data consistently (with a penalty on performance) or eventually consistent. Since SDN applications are OSGi bundles (i.e. applicational containers), data stores are contained within each specific application. The developer is free to expose the datastore directly in the OSGi service directory or to create custom OSGi services that expose behavior rather than data. The later was the selected approach.

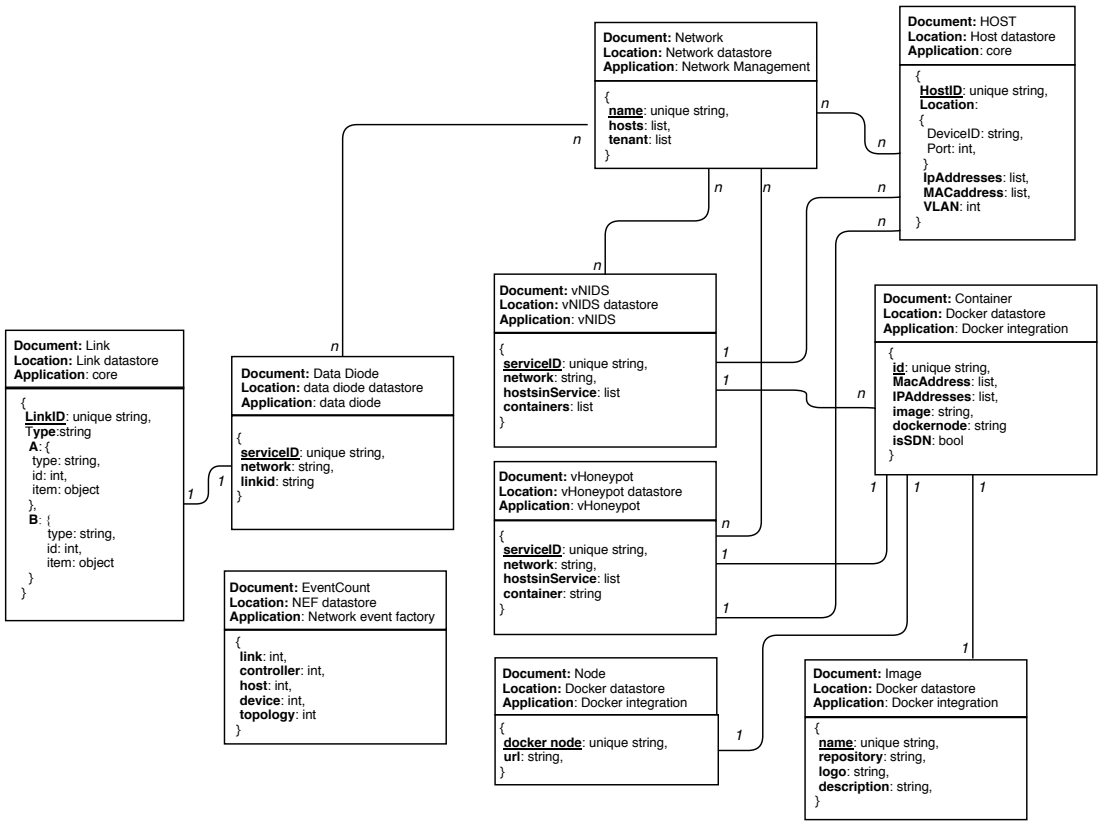


Figure 7.2: Datastore model diagram.

To overcome the limitations of using a non-relational database, each developed application defines Map structures that have a unique primary key (e.g serviceID) and that hold references to unique keys in other documents regardless of its location (e.g. network, container id, etc). The relationship between datastore documents in the IADS subsystem is simplified in Figure 7.2 showing how, ultimately, a "relational model" exists between structures. Almost all application data is stored consistently in the IADS subsystem using the `consistentMap` primitive. The only application using an eventually consistent store is the *Network Event*

Factory application. It simply stores counters for sent events - data that is not relevant enough to justify a performance penalty.

7.4 External interfaces

This subsection details the developed external interfaces (REST, Websockets and command line) of the SDN subsystem. The web-based interfaces are ultimately consumed by the presentation layer of IADS, i.e. the management web-interface, for network orchestration. Command-line provides an easy way for testing and performing operations in the SDN controller itself.

As seen in Section 5.4 the *WebAPI* SDN application is the component responsible for exposing the subsystem functionality over HTTP: defining an abstraction layer that groups the main OSGi services of each of the other applications. This application extends ONOS webserver (Netty) with new HTTP endpoints, following guidelines and good practices for REST API development (Dharani 2017). For each particular applications, the services the application consumes are grouped into a consistent namespace (e.g. networks, docker, vnids) while application objects (e.g. hosts, devices, tenants) constitute a single endpoint. Since HTTP methods map well to CRUD (Create, Read, Update, Delete) operations, different HTTP methods are used on the same endpoint as a means to keep the API simple and intuitive. GET methods often mean the endpoint will return a list of objects, POST methods create new objects, PUT methods update a given object, and DELETE methods delete the object. The REST API returns different HTTP status codes depending on the request. In case of errors, an optional `error` JSON field is returned. Status codes are listed below.

- **200** - Successful request
- **401** - Unauthorized request
- **400** - Bad request *
- **409** - Conflict occurred while performing the request *

The developed REST interface makes use of the *JAX-RS* API (Oracle 2018a) of the included controller Jersey framework (Oracle 2018b), and is fully stateless. Each endpoint expects the presence of an *Authorization* header containing a JSON Web Token (JWT). The token (base64 encoded) is generated by another component of the IADS architecture (IADS-auth, which is out of the scope of this thesis). It includes the role of the user (see Section 4.2.5) and the respective username as the token claims. The *WebAPI* SDN application is able to validate the provided token as it is configured to use the same encryption key as the IADS-auth component. All REST endpoints were tested prior to the web interface development by means of a custom developed python client built around the requests module.

The websockets interface provides an alternative to REST for data that is prone to change periodically, such as the topology graph, the list of running containers or device statistics. Since this data can be changed outside the domain of the management web interface (e.g. by using the controller command-line or interfacing with the tools - OpenvSwitch or docker - directly at the running host) the web interface needs a way to be kept updated without the need of a complete refresh. Similarly to the REST interface, Websockets are secured with the `Sec-Websocket-Protocol` header containing the JWT authentication token. Each

* Error JSON field is returned

websocket instance periodically checks if the connection is still open. Hence, if a given client closes the connection (by leaving a specific view in the web interface) the websocket instance on the server will end up being removed by the socket servlet.

The command-line interface is the least important of all interfaces since it is not directly consumed by any client. It only exists as a means for runtime testing while developing the applications or for the system admin to perform the same operations directly on the controller CLI.

The complete external interface definitions are provided in Annex C of Volume II.

7.5 Virtualization infrastructure

This subsection details the work performed on the virtualization infrastructure. Apart from setting up the system (docker, docker registry and OpenvSwitch installation and configuration), other components had to be built from scratch. Subsection 7.5.1 details the effort in developing the SDN glue agent while Subsection 7.5.2 addresses the development of vProbes.

7.5.1 SDN glue agent

The *SDN glue agent* is an important piece of the overall architecture. Without it, containers deployed from the *Docker integration* application could not be added to the SDN network, vProbe containers would not be recognized as network hosts in the topology graph, and there would be no way to install OpenFlow flow rules at the container boundary. The workflow/design of the agent is represented in Figure 7.3.

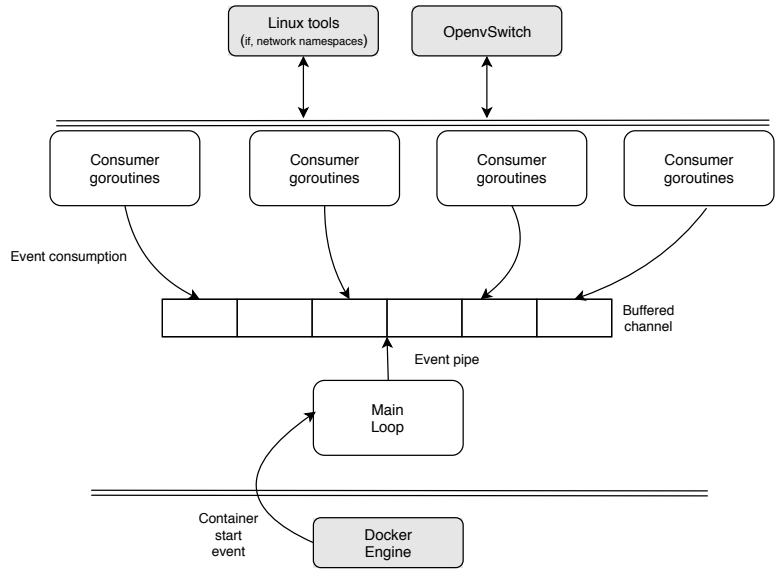


Figure 7.3: SDN glue agent development.

Like the rest of the SDN subsystem, the main quality attributes limiting the agent design are performance and availability. Hence, the agent was developed in the Golang programming language taking advantage of its exceptional concurrency mechanisms (*channels* and *goroutines*). When the agent starts up, a *buffered channel* is created and a number of configurable *goroutines* are launched to consume events published to the channel. The main agent loop listens to docker engine events (using the builtin golang docker library) and simply pipes the events to the buffered channel. Events are consumed (and automatically removed) from the channel by the first *goroutine* to get the event. When an event is consumed (e.g. a container was started), the *goroutine* looks for a label (`-sdn=true`) in the container metadata and attaches the container to OpenvSwitch bridge (in a process identical to the one documented in Section 3.5). The design choice results in a pattern similar to a *publish-subscribe* mechanism within the agent itself. By having several consumers the agent is able to concurrently process multiple events and guarantee containers are added to the OpenFlow network as soon as they are launched.

As part of the development, a *systemd* service was created to start the agent from the command-line. To ease the migration of the agent, it has Makefiles (and a Jenkins CI job definition) to build a CentOS 7 .rpm file out of the agent source code.

7.5.2 Probe development

Probe development was not a direct objective of this thesis. However, to have a working prof-of-concept it was required to also develop or port some software packages to container images. To make the deployment as fast as possible, probes were build using the Alpine Linux base image (only 17 MB per container). The following probes where developed:

- **Snort** (vNIDS) - One of the world's most powerful signature-based network intrusion and detection systems. Highly configurable in what concerns attack detection and preprocessing.
- **Tcpdump** (vNIDS) - A simple probe to capture network packets reaching the SDN interface.
- **Conpot** (vHoneypot) - A known IACS honeypot able to emulate Modbus TCP, Siemens S7 and other SCADA protocols. Custom profiles can be developed on top of the framework and the registries can be customized.
- **Honeyd** (vHoneypot) - A general purpose honeypot to emulate the FTP, SSH, DNS, HTTP, SNMP protocols. Custom configurations are possible to emulate specific operating systems and/or to create custom honeypots.

The developed images were uploaded to the private docker registry of the platform to keep a central vProbe repository.

It is also worth mentioning that the startup of a vProbe container is not as simple as starting the main probe executable. The container has to wait for the SDN to be available (the SDN glue agent has to create the virtual interface and attach it to the OpenvSwitch bridge first) and generate an ARP packet so it can be detected on the controller topology. For this purpose, the `arping` unix tool is used to generate a *gratuitous* ARP request as shown in Algorithm 7.3.

Algorithm 7.3: Probe startup example (not pseudo-code).

```
1  begin
2
3  while !{SDN_INTERFACE}
4    do
5      sleep
6    done
7  arping -U -I {SDN_INTERFACE} {IP_ADDRESS} -c 1;
8  start_probe
9
10 end
```

7.6 Management Web-interface

The ONOS controller already provides a web interface with several functionalities for network monitoring. Nevertheless, a fundamental part of the IADS platform is a custom developed web interface that allows the management and monitoring of all its building blocks. This affects not only the SDN subsystem but all its other components such as probe configuration, streaming platform configuration, domain processor, data lake and SIEM applications (out of the scope of this thesis). As a result, we opted to develop new web views for the SDN subsystem that could seamlessly integrate with the overall IADS web-interface and take advantage of the developed external interfaces (Annex C of Vol. II).

To reduce the effort in developing a new web interface, we only implemented features that did not overlap with the ones already provided by the controller. Functionalities such as listing installed flow rules, network hosts and devices were left out of the web interface. Additionally, an effort was made to keep the web-interface as simple and intuitive as possible, delegating all the fundamental actions to a topology graph view of the network. Note that in IACS, process operators and engineers are used to perform actions at HMI interfaces, unequivocally selecting a given process equipment and checking all the relevant process variables in real time. Similarly, by making the decision of performing all network service's deployment in the topology view, the IADS platform increases its market value by lowering the learning curve in adopting SDN, minimizing the barrier created by the virtualization of its probes.

The implemented user interface is thus responsible for network topology operations (service deployment and network segmentation), virtual infrastructure configuration (adding or removing virtualization nodes) and vProbe monitoring. User interface elements are correctly filtered according to the degree of permissions of the logged user. Figure 7.4 shows the network topology view of the IADS web interface, along with the SDN subsystem menu. It is possible to see that the main options accessible via the web interface are:

1. Main menu.

2. Live topology graph.

3. Topology actions menu.

4. Network graph filtering by logical sub-network.

5. Information panel per selected topology asset.

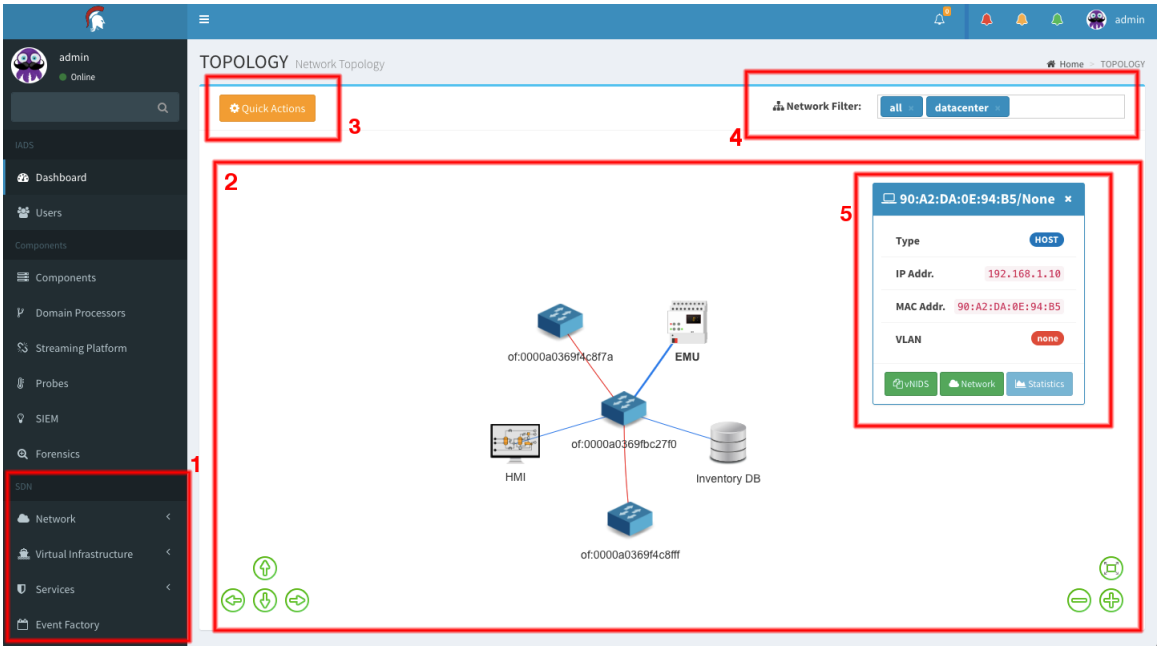


Figure 7.4: Main SDN subsystem view.

The interface main menu has the following options:

- Network (topology and network list).
- Virtual infrastructure (node and registry configuration; vProbe listing and monitoring).
- Services (list vNIDS, vHoneypot and data diode services).
- Event factory (enable and access SDN event piping to IADS domain processor).

Regarding the network topology view, selecting an asset will show different information and options depending on its type (host, vProbe, switch or link), as shown in Figure 7.5.

Figure 7.5: Information panel depending on the asset type.

By clicking on the filtering select element, the topology graph is filtered in realtime, showing the combined graph of the selected logical networks. Figure 7.6 shows the difference in

selecting only the *datacenter* network and the *all* option (two new switches appear that correspond to the two virtual infrastructure switches).

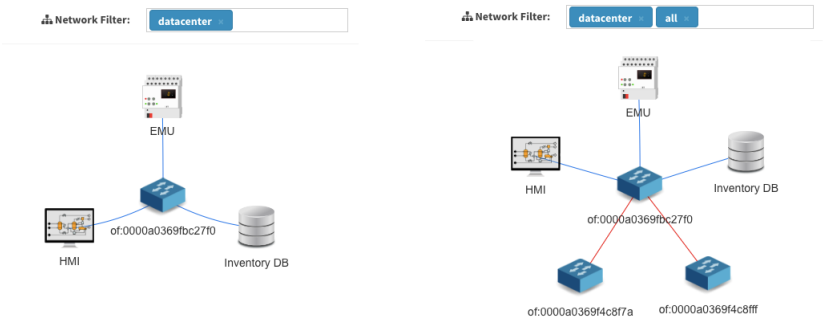


Figure 7.6: Network graph filtering.

The *quick actions* button in Figure 7.4 gives access to the main operations of the IADS subsystem, namely:

1. Create network.
2. Associate a network tenant with a network.
3. Deploy a service.

by opening an overlay over the topology (Figure 7.8).

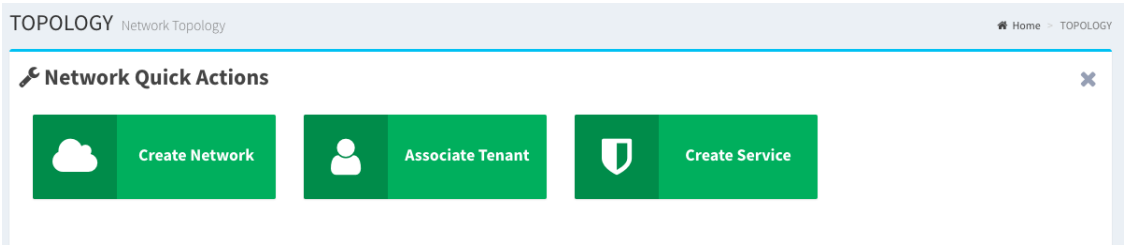


Figure 7.7: Topology quick actions.

Creating a network or associating an existing network with a tenant profile are simple modal dialogs with input and dropdown elements. Hosts are added to an existing logical network by clicking on the *network* button in the info panel of the host. By clicking on *services* in Figure 7.8, the user is presented with several container images and associated service.

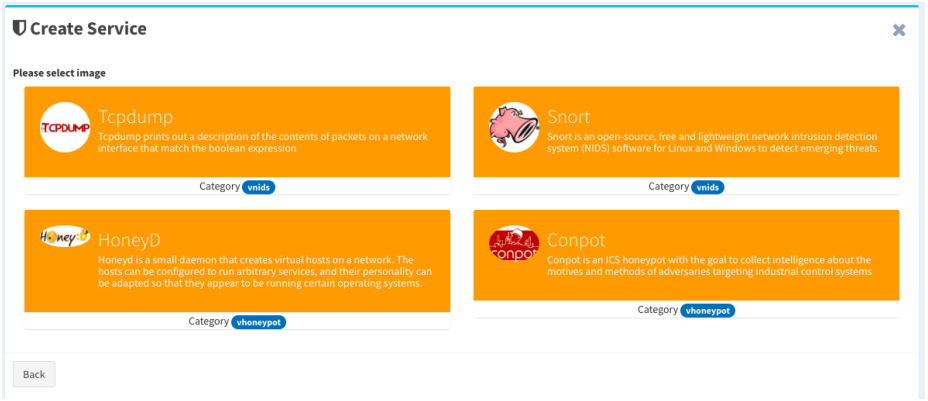


Figure 7.8: Services available for vProbe deployment.

Further configurations are available after selecting an image. For example, configurations pre-vHoneypot deployment are shown below.

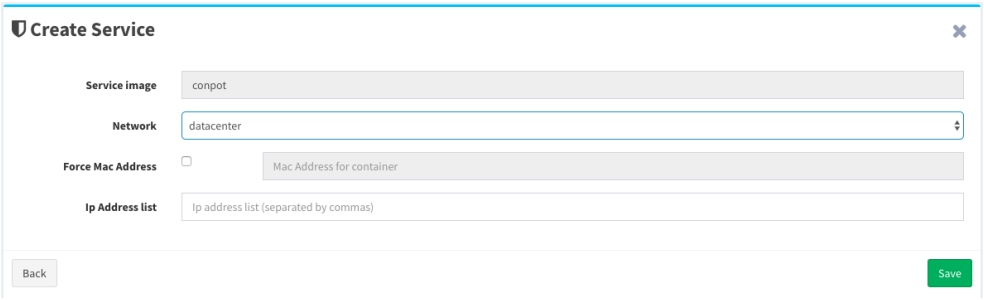


Figure 7.9: Additional configurations for vHoneypot.

Hosts are added to a created vNIDS service by clicking on the *vNIDS* button in the information panel of an host. Similarly, a data diode is deployed by clicking on the *data diode* button on the link infopanel. Figure 7.10 shows the topology view with an honeypot, vNIDS and data diode deployed as seen in the webinterface with clear visual feedback provided to the user.

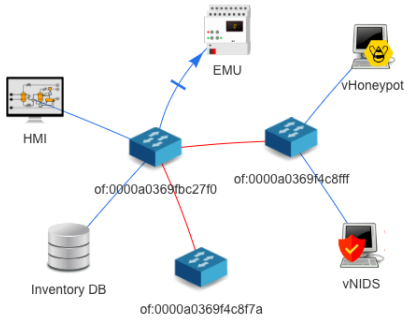


Figure 7.10: All services deployed on a logical network.

The virtualization infrastructure menu item allows adding or removing virtualization nodes from the platform (c.f Figure 7.11).

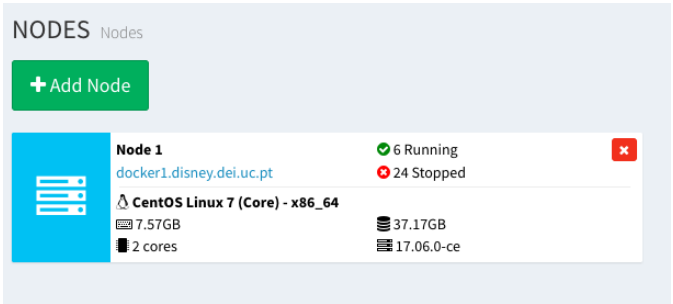


Figure 7.11: Add or remove virtualization nodes from the platform.

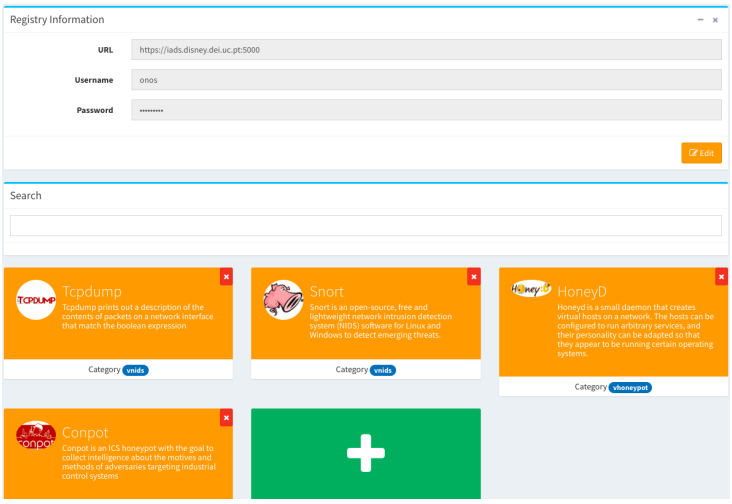


Figure 7.12: Configure the image registry (vNIDS and vHoneypot).

Adding new images to the registry (or modifying the registry information) is also possible, making the overall virtualization infrastructure to work similarly to a vProbe security store - as shown in Figure 7.12.

It is also important to mention that statistics concerning deployed containers (those of vProbes) are available at a specific view. It can be accessed via the list of deployed services or in the information panel that opens after clicking on a vProbe host. In this view, CPU and RAM usage can be checked in real-time and network statistics (byte and packet counters) are displayed in the form of live charts. In the same view a console of the container (and their logs) is also presented as illustrated in the figure below.

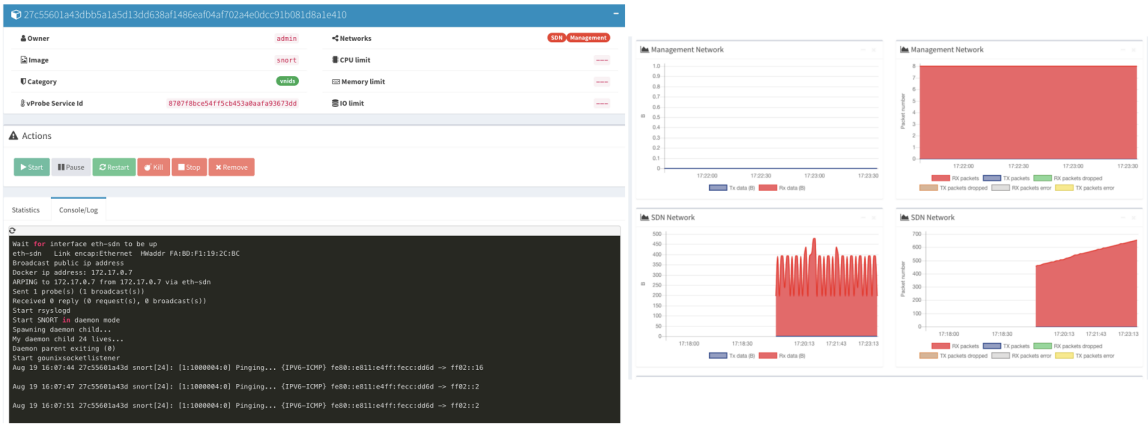


Figure 7.13: Detailed view of a vProbe container with its network statistics and console.

The web-interface of IADS tries to respect the Norman principles for interaction design (Norman 2013) as much as possible. Feedback for any executed action is provided through alerts in the user interface. Furthermore, the execution of operations on the topology graph helps the user to develop a mental modal regarding the use of the web-interface.

The developed interface is written in Javascript, HTML and CSS, using the Vue JS framework and webpack for artifact generation. It makes uses of the *Vuex* module (Vuex 2018)

for global state sharing between components and the *Axios* module (Axios 2018) to perform promise-based HTTP requests.

7.7 Chapter wrap-up

Globally, no subjective choice was made on the selected tools, frameworks and programming languages. They were selected either due to imposed limitations of the reused software packages or to "maximize" the quality attributes (performance and availability) that are requirements of both the platform and its applied context. This approach was followed even at the cost of a steeper learning curve. The chosen approach to flow rule installation (and application algorithms) was also made to overcome the complete dependency on the control plane and to take advantage of the distributed nature of the control plane - distinguishing the work of this thesis from the majority of the state-of-the-art in the field. The web-interface was planned with focus on the final user - critical infrastructure operators - borrowing ideas from similar human-machine interfaces they are used to work with and prepared for multi-tenancy with role segregation. The developed subsystem integrates seamlessly in the IADS platform, resulting in a powerful framework, capable of being used for research in the future (for a broad range of fields/use cases) while using a scalable, flexible and decentralized infrastructure.

Chapter 8

Validation

This chapter is focused on the validation of the developed framework and overall SDN subsystem. Section 8.1 explains the testbed layout specifies all the machines used in the validation scenarios. Section 8.2 focuses on the functional validation with a set of use cases created to evaluate each specific application. Section 8.3 details the tests executed to access the non-functional aspects of the platform with focus on its performance, scalability and availability. Finally, section 8.4 provides a chapter wrap-up, drawing the main conclusions from the validation scenarios. Note that the functional validation of the subsystem was verified by tracing the elicited use-cases to the implemented functionalities, as referred in Chapter 6.

8.1 Testbed description

Figure 8.1 shows the testbed implemented to demonstrate and validate the proposed SDN subsystem. The control plane was composed of a cluster of containerized ONOS controller nodes (scalable from 3 to 5 nodes), each one running on a different VM. The hypervisor containing the VMs had 192 GB of RAM and 2 physical Intel(R) Xeon(R) CPUs @ 2.80GHz each. Each controller VM was assigned 30GB of RAM and 8 vCPUs. A Dell Poweredge R210 server with 8 GB of RAM and running CentOS 7 was configured to work as an OpenFlow switch. The server had five gigabit Ethernet NICs and the OpenvSwitch (open-source software switch implementation) installed. Furthermore, OpenvSwitch was configured with Intel DPDK (TLF 2018) for increased network performance (see Section 7.1).

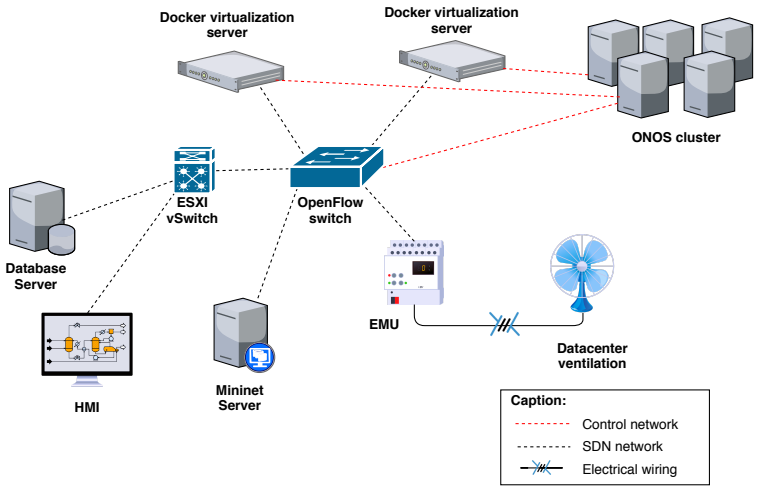


Figure 8.1: Testbed scenario.

Two virtualization servers, following the architecture illustrated in Figure 5.18 were directly attached to the physical OpenvSwitch, to act as virtualization nodes for the deployment of vProbe containers. Both virtualization servers were commodity HP Proliant servers (contemporary Core 2 Duo Xeons) with 8 GB of RAM, with the Docker engine and an OpenvSwitch installed. The servers contained two interface cards: one for direct physical attachment to the OpenFlow switch and another to access the management network (for container deployment).

In what concerns the network hosts, the implemented testbed was composed of four different machines. Two ESXi VMs served the purpose of simulating a database server and an Human Machine Interface (HMI). Both VMs had virtual NICs attached to a ESXi vSwitch configured in passthrough mode in respect to the physical server NIC (so the vSwitch itself did not represent a non-Openflow layer in the overall topology). The HMI machine was running RapidScada, issuing queries to the topology Environmental Monitoring Unit (EMU) in order to present the datacenter humidity and temperature values in a user-friendly interface. The HMI monitored both values and triggered the start of a ventilation fan if the temperature exceeded a certain threshold. Temperature and humidity values were kept updated by the EMU in three holding registries and made available to the SDN network via the Modbus TCP protocol.

A Mininet VM was also attached to the main OpenFlow switch as a means to evaluate the scalability of the system. Mininet is an Ubuntu based distribution that allows the creation of virtual OpenFlow switches and network hosts, using Linux control groups and network namespaces. With Mininet it was possible to create hybrid network topologies with a part being represented by virtual network assets (in custom-"scripted" topologies). The server had a total of 16GB of RAM and ran in a Citrix XEN server hypervisor.

8.2 Functional validation

A set of scenarios was defined to functionally validate the developed platform. The tests herein documented were performed in the testbed illustrated in Section 8.1, and tried to use tools from the IACS domain whenever possible or justified.

8.2.1 vNIDS evaluation

The functional validation of a vNIDS deployment took place in November 2017, at the European Commission, during the intermediate review of the ATENA project. The security use case demonstration was based on a situation where the Inventory database host was compromised by an external attacker which was able to gain access to the host command line. A logical network was previously created and the three hosts (EMU, HMI and InventoryDB) were added to it so they could have connectivity. A TCP SYN-flood attack was launched from the database server against the EMU in order to "blind" the HMI. Under the attack conditions, the HMI was no longer capable of performing the Modbus TCP queries for temperature and humidity nor to obtain the state of the fan. The temperature of the datacenter was increased with the help of a heater to confirm the EMU would not start the fan (Figure 8.2).

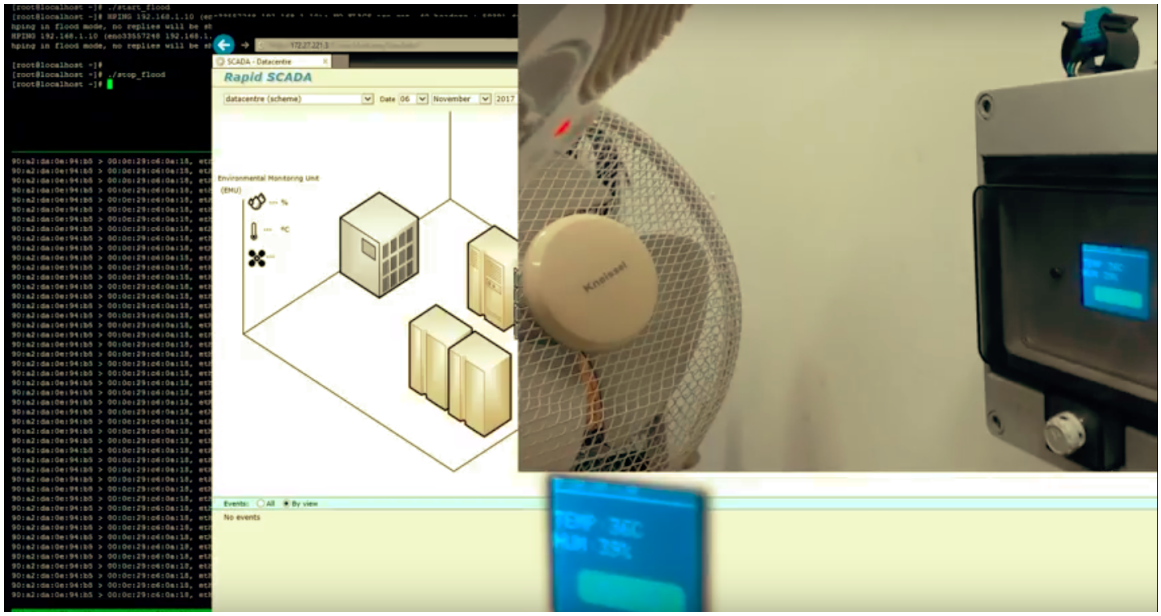


Figure 8.2: TCP flood attack against the EMU and the HMI unable to trace the operational variables.

The infrastructure operator, accessing the HMI would not have any hint that a cyber-attack was targeting the datacenter network. The same attack was replayed after deploying a vNIDS from the IADS web-interface: a vNIDS service was created, the container image was selected (a Snort based image with pre-configured rules to detect SYN-flood attacks) and all the hosts in the datacenter logical network were associated with the service. The association of hosts to the service led to the deployment/launch of a new vProbe container, located at one (arbitrary) virtualization node of the testbed (Figure 8.3).

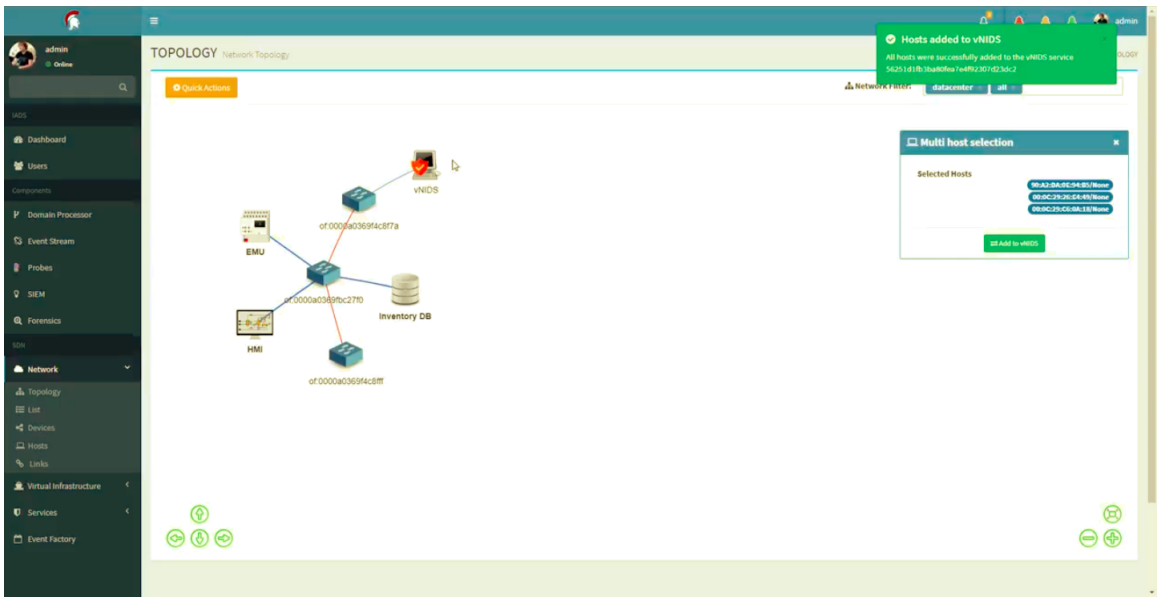


Figure 8.3: vNIDS launch.

By replaying the attack with the vProbe deployed, the underlying switch fabric made copies

of the network traffic available to the vProbe container. This triggered the signature-based rule installed in Snort and the container was able to generate, encode and send an event to the upper layers of the IADS architecture. Ultimately, the event reached the IADS web interface as shown in Figure 8.4.

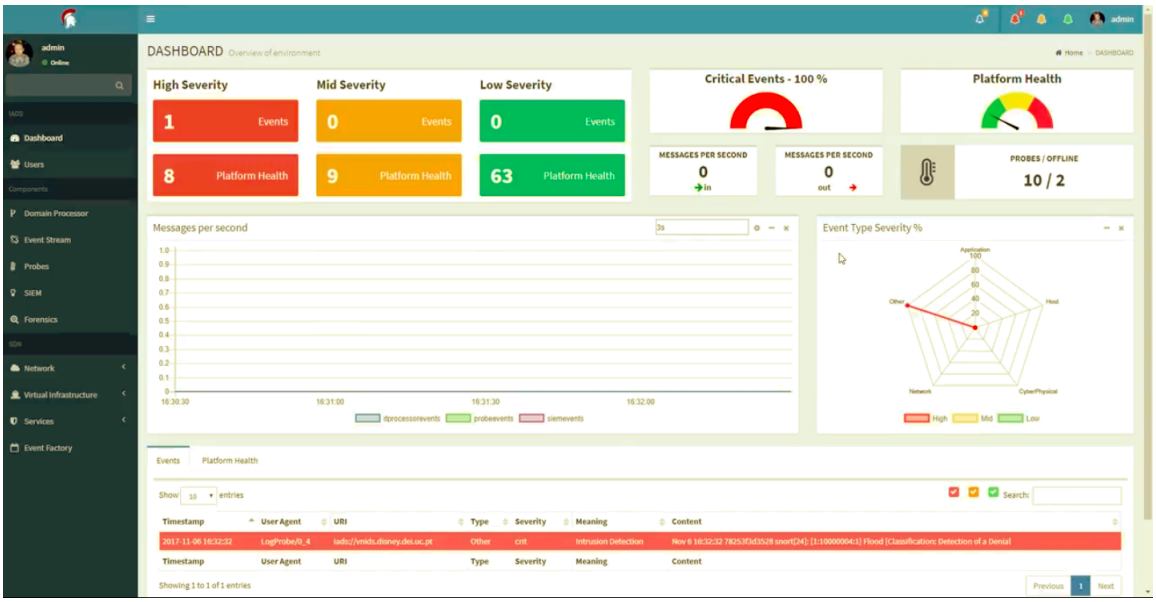


Figure 8.4: IADS denial-of-service alert issued by the vNIDS probe container.

This use-case proved the suitability of the SDN subsystem to start containers on the virtualization infrastructure and to program the underlying network to duplicate network traffic with the launched container as destination. Videos for this particular use case demonstration are publicly available in Youtube (ATENA youtube 2018) as part of the intermediate project review.

8.2.2 vHoneypot evaluation

The vHoneypot functional validation was twofold, since each of the developed vprobes (conpot and honeyd) have different purposes and allow the evaluation of different functionalities of the vHoneypot SDN application. In both cases, similarly to Section 8.2.1, it was assumed that from a security standpoint the InventoryDB host machine was compromised by an external attacker. Using the IADS web-interface (network topology view), a vHoneypot service was created on the *datacenter* logical network and Conpot was selected as the container template image. The default conpot profile emulates a Siemens SIMATIC S7-200 PLC with S7comm, Modbus TCP and SNMP support. This process resulted in the automatic deployment of a vHoneypot container with the IP address `192.168.5.10` attributed by the DHCP SDN application (Figure 8.5). From the InventoryDB host terminal, the PLCScan tool (PLCScan 2012) was used to enumerate any existing PLCs at the given IP address, as shown in Figure 8.6.

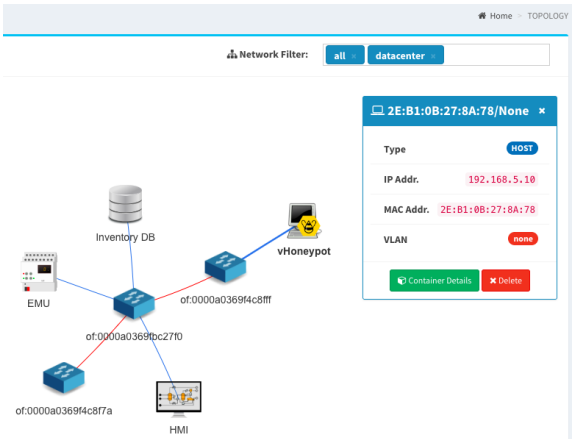


Figure 8.5: vHoneypot container deployed in the datacenter logical network (plus network information).

```
[root@attacker plcscan]# python plcscan.py 192.168.5.10
Scan start...
192.168.5.10:102 S7comm (src_tsap=0x100, dst_tsap=0x102)
  Module                 :  v.0.0
0000000000000000)
  Name of the PLC        : Technodrome
00000000000000000000000000000000)
  Name of the module     : Siemens, SIMATIC, S7-200
53372d3230300000000000000000)
  Plant identification   : Mouser Factory
00000000000000000000000000000000)
  Copyright              : Original Siemens Equipment
717569706d656e74000000000000)
  Serial number of module  : 88111222
00000000000000000000000000000000)
  Module type name       : IM151-8 PN/DP CPU
00000000000000000000000000000000)
  OEM ID of a module     :
00000000000000000000000000000000)
  Location designation of a module:
0000000000000000000000000000000000000000)
192.168.5.10:502 Modbus/TCP
```

Figure 8.6: PLCScan results against the vHoneypot container.

The scanning process against the vHoneypot container IP address found a Siemens PLC as expected. This proved the vHoneypot SDN application was able to correctly program the underlying network to enable the conpot operation.

A subsequent test consisted on the creation of another vHoneypot service on the datacenter network by:

1. Selecting the Honeyd container image from the private registry.
2. Setting the option to reserve the fake ip `192.168.1.137` to the vhoneypot service.

The honeyd container image had a Windows NT profile mapped to the "fake" IP address and the following services enabled: SSH, telnet, HTTP server and DNS server. Once the service was created, the resulting container got the IP address `192.168.6.88` assigned by the DHCP application. The Nmap tool was used to perform a port-scan against the "fake" IP address. Figure 8.8 shows the scanning result.

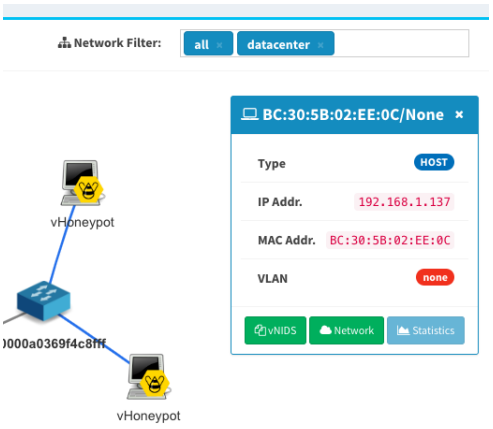


Figure 8.7: Network topology graph of the IADS web interface showing 2 hosts for the same container.

```
[root@attacker ~]# nmap 192.168.1.137

Starting Nmap 6.40 ( http://nmap.org ) at 2018-08-11 19:02 BST
Nmap scan report for 192.168.1.137
Host is up (0.026s latency).
Not shown: 997 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
23/tcp open  telnet
80/tcp open  http
MAC Address: BC:30:5B:02:EE:0C (Dell)

Nmap done: 1 IP address (1 host up) scanned in 0.55 seconds
[root@attacker ~]#
```

Figure 8.8: Nmap port scan against the fake IP.

Results shown the attacker was able to correctly enumerate the container emulated services even though it was scanning a non-existing IP address. In fact, after the scanning process, a new host with the IP `192.168.1.137` appeared on the topology - the result of faked ARP packets generated by the container.

8.2.3 Data diode evaluation

For validating the data diode service operation, we assumed the InventoryDB and Mininet machines played the role of usual application proxies and protocol breakers (RX and TX agents) in traditional data diodes (see companion research paper in Annex D of Vol.II). A data diode was deployed in the link between the RX and the testbed OpenFlow switch (in the host direction). Hence, although there is a logical network between the three hosts, the link between the RX agent and the rest of the network was effectively unidirectional (receiving only). Figure 8.9 shows the real IADS web-interface topology view, while Figure 8.10 presents a diagram representation of the test.

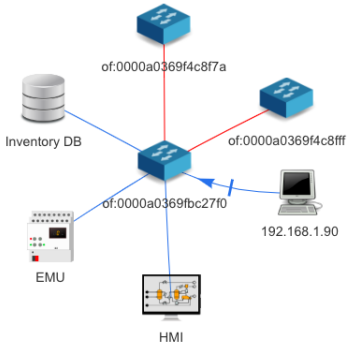


Figure 8.9: IADS web-interface topology graph after the deployment of the data diode.

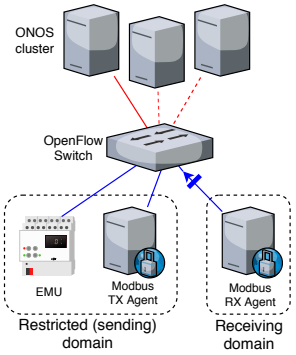


Figure 8.10: Representation of the restricted connection domains and the unidirectional link.

Recurring to the Netcat tool (Giacobbi 2006), the RX agent was configured as an UDP server while the TX agent acted as a client, and vice-versa. We confirmed that in the former case packets were able to flow while in the last no communication occurred.

Using a modified version of the Dyode framework (Wavestone 2018), a test was designed to simulate a real industrial control system operation aiming at studying the effectiveness of data diode in supporting bidirectional protocols (Modbus TCP) and the latency created by such process. In this test, the TX agent queries the EMU holding registries, serializes the data into the pickle format and sends it through the UDP protocol to the RX agent. The RX agent behaves as the EMU device on the other side of the network: it deserializes the received data, updates the internal registries and exposes a Modbus TCP server. An increasing number of sequential reads of ten EMU holding registries was then performed. To accurately collect the time values we removed the ability to process and packetize the obtained data from the TX agent and measured the time immediately before and after each query. The measured times should be taken as the base values for reading latency. For the RX readings, the time was recorded right after data has been deserialized and updated in the agent context. Moreover, a counter was increased upon receiving a reading from the TX agent. Total test duration was computed using the temporal instant before the first

query and both machines were synchronized via NTP. The results (5 sample test) can be found on Table 8.1.

Table 8.1: Latency effect of the data layer on Modbus TCP readings.

Modbus Agent	Number of Queries	Time (s)	Failed Reads (%)
TX	1	0.067 ± 0.139	-
	10	9.889 ± 0.640	-
	100	111.045 ± 0.331	-
	500	566.654 ± 0.558	-
RX	1	0.654 ± 0.344	0
	10	10.185 ± 0.777	0
	100	111.820 ± 0.897	0
	500	567.679 ± 0.549	0.360 ± 0.444

The obtained results allow to conclude the same functionalities available in *commercial-off-the-shelf* data diodes are easily implemented in software, using an SDN approach. Despite all the additional processing work performed by both proxy agents, the induced latency of the Modbus readings was only one additional second for the highest number of sequential reads. However, note that as the number of reading operations increases, a few failed reads were noticed. This is expected due to the no-guarantee nature of the UDP protocol. This issue is further explored in Section 8.3.1.

8.2.4 Network event factory evaluation

The network event factory application was validated by simply leaving the system configured to send all SDN events to a known Kafka broker and topic for a long period of time. After 2 days of stable operation, the SDN subsystem was able to pipe more than 385,000 events to the upper layers of the IADS platform (Figure 8.11). The events were confirmed received by inspecting the respective Kafka topic.

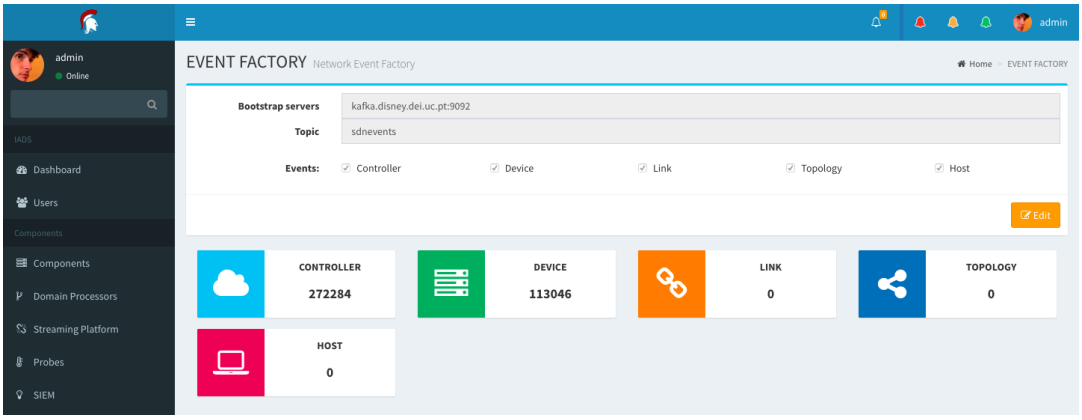


Figure 8.11: Network Event Factory application test.

The absence of *Link*, *Topology* and *Host* events generated during this period also serves to prove the stability of both the testbed and the overall IADS SDN-subsystem. Unlike device and controller events (which provide statistic information every 5 seconds), those events are only generated if changes occur in the topology.

8.3 Non-functional validation

This section provides the non-functional assessment of the SDN subsystem. All the tests were carefully designed in order to reason and conclude about the main quality attributes chosen for the platform: performance and scalability (Section 8.3.1), and availability (Section 8.3.2).

The containerization of the control plane, although contributing to the overall portability of the system, poses numerous challenges to the correct collection of time-based metrics. With the OpenFlow protocol, any action taken by the controller in respect to a specific network event always starts with a *PACKET_IN* message sent by an OpenFlow switch to the controller node which has the mastership of the switch. After processing the network packet, the SDN controller node synchronizes its state with all the other nodes in the cluster and generates an event that can be consumed by the controller application layer. For some tests, to accurately compute the latency of a given action both the Openflow network packets and the controller events need to intercepted. Network packets arrive at the physical host where the controller node is running. SDN controller events are triggered to SDN applications that run in the controller node (in an OSGi environment sand-boxed in a Docker container). This makes the non-functional validation an extremely complex process – requiring means of automation and mechanisms for inter-process communication.

As a result, an SDN application was developed (and installed in the controller cluster) to collect all the meaningful network events. Concomitantly, a small program was developed and installed in the controller host (outside the container) to intercept and inspect network packets arriving at the node. The overall workflow followed to perform the non-functional validation is presented in Figure 8.12.

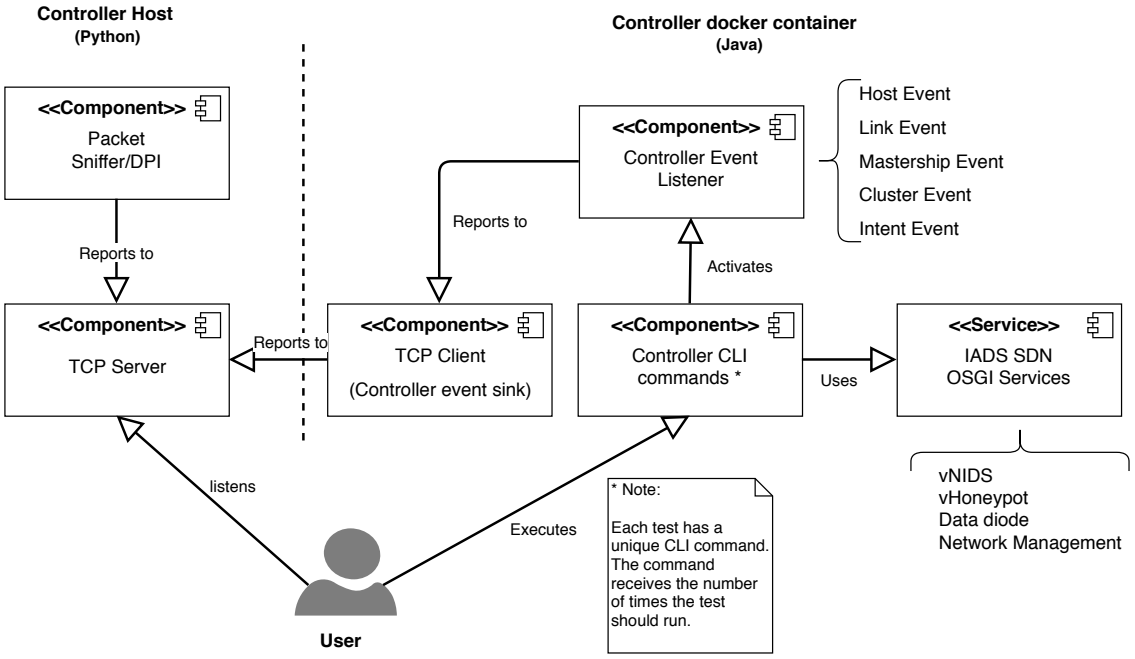


Figure 8.12: Non-functional validation workflow.

For each test, a controller CLI command was developed to automate and repeat each test a given number of times. Those CLI commands activate specific *Event Listeners* (depending

on the test) and consume the logic exposed by the OSGi services developed as part of the IADS subsystem. When a given event is triggered by the controller cluster, the *Event Listener* annotates the event with a timestamp and pipes it to a common observable component which acts as a sink and dispatcher for events. This component is in fact a TCP client, connected to a TCP server running on the host (via the *loopback* interface) which simply flushes the received events to the *standard output*. The TCP server also dispatches a *Packet Sniffer* thread that intercepts any *PACKET_IN* OpenFlow packets, processes and filters each packet according to a given rule (e.g. ARP packet). The sniffer made use of the much known *Scapy* Python library (Biondi 2018). It is worth mentioning that although there is latency in the SDN Application → TCP Server channel, it does not reflect on the collected values. Measured latencies use both the timestamps of the intercepted network packets (provided by Scapy) and the timestamps of the received SDN events, "tagged" by the SDN validation application right after the event is received. To execute a given test, it is required to start the TCP server on the Host and execute the specific test CLI command on the controller CLI. Confidence intervals for all test results were computed using a normal distribution with 95% of confidence level, unless stated otherwise.

8.3.1 Scalability and Performance

The following subsections detail all the non-functional tests performed to access the performance and scalability of the SDN subsystem. Some of the tests attempt to evaluate features that are either built into the chosen distributed controller (ONOS) or that depend on the OpenFlow switch software (OpenvSwitch). They do not have a strict dependency on the developed applications but still affect the performance of the subsystem as a whole.

Host detection

Host detection is an important part of the IADS subsystem. Although it is part of the controller core functionalities, it strongly affects any of the developed applications. To be able to create a logical network via the developed *Network management* application, to add a specific network host to a vNIDS, to deploy a data diode or vHoneypot in a given network, the controller has to know the host beforehand. As seen in Section 7.2, it is upon the reception of an ARP packet sent by a network host that the controller computes the topology graph and fills the Host internal datastore. Hence, minimal latencies when detecting network hosts are desired.

The test procedure was as follows:

1. The TCP server was started on the network controller host and the test CLI command was started in the controller CLI (configured to wait for 25 events).
2. From an host connected to the physical switch of the testbed (see Figure 8.1), ARP packets were generated 25 times using the *macof* utility.
3. The host detection latency was calculated as the difference between the timestamp of the ARP *PACKET_IN* and the timestamp of the *HOST_ADDED* controller event.

The independent variables for the test were respectively:

- Network controller cluster size (1, 3 and 5 nodes).

The results of the test are ploted in Figure 8.13 and summarized in Table 8.2.

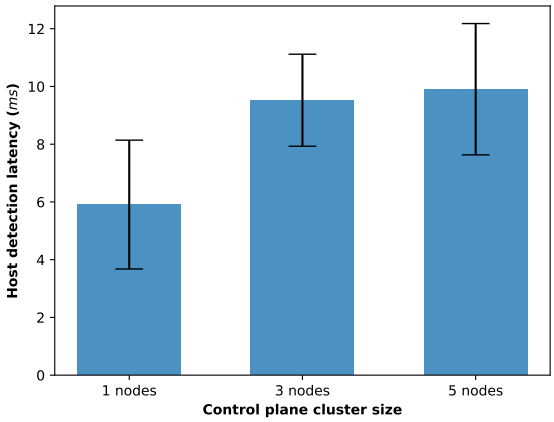


Figure 8.13: Host detection latency depending on the control plane cluster size.

Table 8.2: Host detection latency depending on the control plane cluster size.

Number of controller nodes	Host detection latency (ms)
1	5.909 ± 2.231
3	9.524 ± 1.594
5	9.905 ± 2.274

By analyzing Figure 8.13 it is possible to conclude that the formation of a SDN controller cluster contributes negatively to the host detection latency. This is expected since the *HOST_ADDED* event is only propagated to the validation application after the cluster state is synchronized. It is also possible to see that increasing the cluster size (from 3 to 5 nodes) does not play a significant effect on the obtained latency. The latency for a 5-node cluster has a mean value higher (but not too distant) from the one obtained for the 3-node cluster although with higher variance. Despite the effect of the cluster size, the control plane shows small latency values for the detection and addition of new hosts to the topology, ranging from 5 to 9 ms.

Network topology scaling

The network topology scaling test aimed at studying the effect of the number of connected OpenFlow switches on the controller global topology graph construction. Furthermore, it also helps understanding the effect of the control plane cluster size on the said process and the respective implications of its built-in mastership load balancer. The test followed a different process than the one detailed in Figure 8.12 - it was conducted solely in Mininet. A Mininet script was created to generate a growing number of virtual switches (from 10 to 1,000) and each switch was associated with the controller cluster. Virtual switches used one node as master (active connection) and all the others as slave/redundant connections. Timestamps were collected before and after running the main virtual switch start loop.

The independent variables of the test were respectively:

- The number of switches (10-1,000),
- The control plane cluster size (1, 3 and 5 nodes).

For each combination, the test was repeated 25 times. Mean values can be seen in Figure 8.14, while Table 8.3 summarizes all measurements. Note that in this test the results strongly depend on the performance of Mininet and are constrained by the Mininet machine resources. Nevertheless, the test allows to infer the ONOS ability to handle complex topologies and manage switch associations.

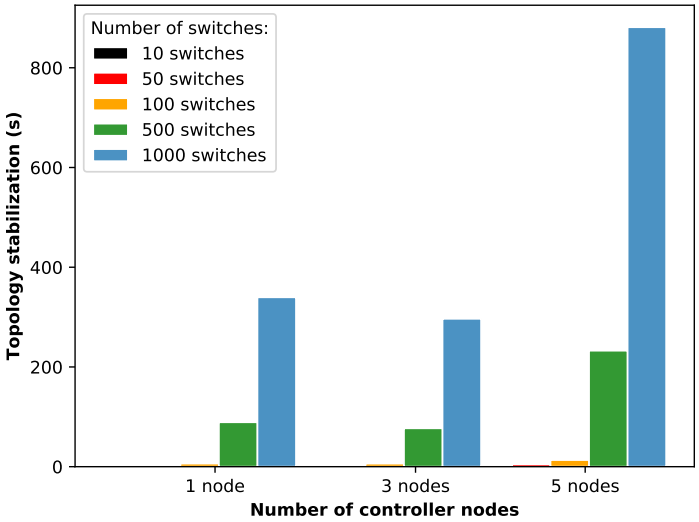


Figure 8.14: Topology scaling test results - the effect of the control plane size and network complexity on the topology construction.

Contrarily to what was expected, Figure 8.14 shows that scaling the control cluster from 1 to 3 nodes does not play a significant difference on the obtained timings. For the most complex networks (500 and 1,000 switches) the process was even faster for the cluster than relying on a single-node controller. This fact shows that apart from the increased control plane availability, the use of a control node cluster can improve the overall system performance since the cluster is able to balance the mastership of each switch across all available nodes. However, with the biggest cluster size (5 nodes), the overall topology creation was slower. This can be explained by the fact that when a new switch attempts to establish a connection with the cluster the controller automatically tries to modify the mastership of the switch so that all nodes are masters of an equal number of switches. This means the controller can change the original master of the switch and also reassign other switches previously associated with the cluster. For a 5-node cluster, the controller has more nodes to chose from when performing the association but it also has a higher number of nodes to equalize. Furthermore, the fact that all switches are virtualized on the same machine (due to the usage of Mininet) also affects the overall test results.

Table 8.3: Topology scaling results.

Cluster size	Number of switches	Topology stabilization (s)
1 node	10	0.41 ± 0.002
	50	2.461 ± 0.01
	100	6.212 ± 0.009
	500	89.13 ± 0.756
	1000	339.719 ± 0.604
3 nodes	10	0.443 ± 0.014
	50	2.579 ± 0.064
	100	6.159 ± 0.074
	500	76.998 ± 0.17
	1000	296.597 ± 0.396
	10	0.737 ± 0.01
	50	4.863 ± 0.024

5 nodes	100	13.325 ± 0.265
	500	232.658 ± 1.229
	1000	881.347 ± 1.208

Despite the issues mentioned above, the testbed control plane cluster was able to keep a stable operation, regardless of the massive amount of connected switches. Note that in IACS, although network topologies are composed of a high number of switches with low port count (4 to 8 ports), the test maximum value of 1,000 switches is an extreme value compared to the number of switches typically found in process control. Moreover, IACS network topologies are quite predictable (and stable) which means the topology graph theoretically only needs to be generated once.

Logical-network creation

The logical-network creation test was designed as a means to evaluate the performance of the developed *Network Management* application, i.e the SDN application responsible for the creation of network slices in the overall topology graph and that provides the core for multi-tenancy in the IADS platform. Network topologies were generated in Mininet and associated with the controller cluster. Spine-leaf topologies were scripted assumed 8 port switches for leaf switches and a maximum of 16 ports for spine switches (see Figure 8.15).

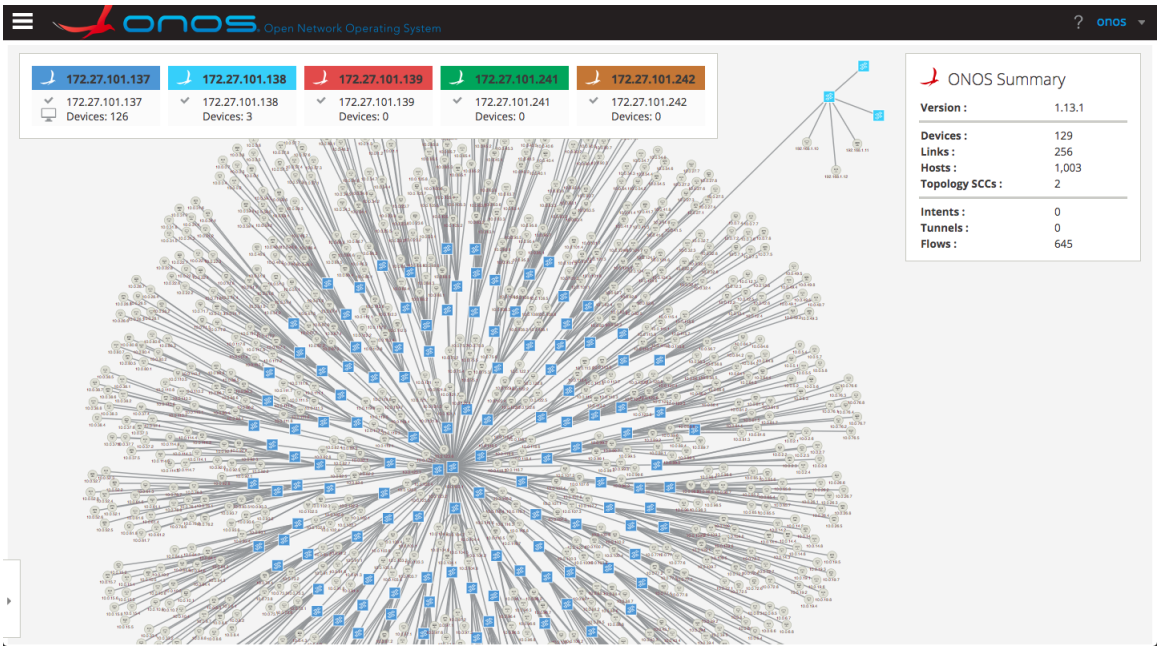


Figure 8.15: Spine-leaf topology example in the ONOS web-interface.

This kind of topology was chosen to emulate a real process control network.
Please recall that logical networks are created pro-actively using the ONOS intent framework by installing *Host-to-Host* intents between host pairs (and are ultimately translated into flow rules). When the number of hosts to be added to a network scales, the number of installed intents and flow rules grows exponentially. This can represent a problem since switches normally have a maximum *TCAM* capacity of about 15,000 rules. Hence, the topologies

generated for this test had a maximum number of 100 hosts. Table 8.4 summarizes the network size (number of hosts) used in the test along with the number of intents and flow rules that are installed in the switch fabric when creating a network with the specified size.

Table 8.4: Number of intents and installed flow rules depending on the number of hosts of the network to be created.

Number of Hosts	Number of Intents	Number of installed flow rules
3	3	18
10	153	720
50	1,653	9,184
100	5,356	30,765

The test was executed in the controller CLI (see Figure 8.12) through a specifically developed CLI command that used the exposed *Network Management* OSGi service to create and remove networks in a loop. Each test was executed 25 times varying the network size and control plane cluster size. Results are presented in Figure 8.16, while the overall data can be found in Table 8.5.

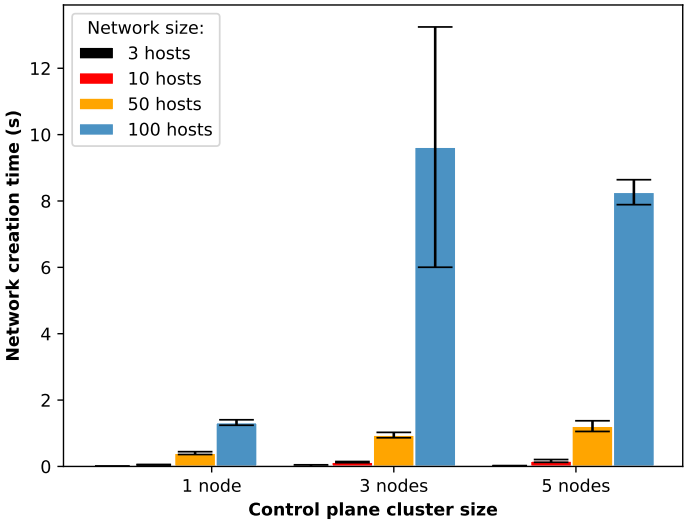


Figure 8.16: Logical network creation times depending on the control plane cluster size and number of hosts.

The obtained results show that even in the case of a 100 host network, in which more than 30,000 flow rules are installed, the network creation process has minimal latency - with mean values between 8 and 10 seconds. The obtained values are in line with the test outlined in (ONOS testcases 2018), which obtained a 2 to 3 second latency per 1,000 intent batch install.

Table 8.5: Logical network creation times depending on the control plane cluster size and number of hosts.

Cluster size	Network size	Network creation time (s)
1 node	3 hosts	0.007 ± 0.001
	10 hosts	0.051 ± 0.009
	50 hosts	0.402 ± 0.042
	100 hosts	1.324 ± 0.082
3 nodes	3 hosts	0.024 ± 0.025
	10 hosts	0.131 ± 0.018
	50 hosts	0.946 ± 0.081
	100 hosts	9.624 ± 3.621
5 nodes	3 hosts	0.021 ± 0.003
	10 hosts	0.166 ± 0.042
	50 hosts	1.215 ± 0.161
	100 hosts	8.265 ± 0.376

It is also worth noticing the high increase in the logical network setup time when we move from a single-node controller to a cluster installation. For the 100 hosts network, the latency increases by a factor of 4 when a cluster is formed - which is expected due to cluster synchronization. Some peak values were also seen in the experiment with a 3 node cluster (reflected on the high variance values). This indicates that forming a bigger cluster is advantageous to reduce the number of masterships attributed to each controller node and to distribute flow rule installation across them. Note that each switch periodically reports the OpenFlow counters per switch port and flow rule to the respective master node. If the number of installed flow rules is high, a controlled node can be flooded when the periodic reporting occurs.

Virtual probe deployment

With the networks previously created (c.f. Section 8.3.1), virtual probes (vNIDS and vHoneypot containers) were deployed 25 times in a loop. Virtual probes are deployed in the virtualization infrastructure physical servers (see Figure 5.20) and a path is required between the emulated Mininet hosts and the actual deployed vProbe container. Hence, there was the need to create an hybrid topology. To accomplish this, one of the physical ports of the Mininet machine (physically connected to testbed OpenvSwitch) was added to one of the Mininet OpenFlow bridges. As a result, the vProbe container was able to receive network packets originated in the emulated hosts. In Figure 8.15 it is possible to see the physical testbed on the top right corner.

Elapsed times were collected before and after each individual step performed by the vNIDS and the vHoneypot applications during a probe deployment (container deployment, DHCP container assignment and network programming). Figure 8.17 shows the relative proportion of each individual step. It is possible to see the developed application is able to deploy a virtual probe in approximately 4.5 seconds. This time is almost completely devoted to the docker container startup/launch. For a 3 host network, the network programmability is almost negligible. Figure 8.18 shows the container startup depending on the probe image for three of the implemented probes. It is possible to conclude there are no significant differences in the startup times of each container.

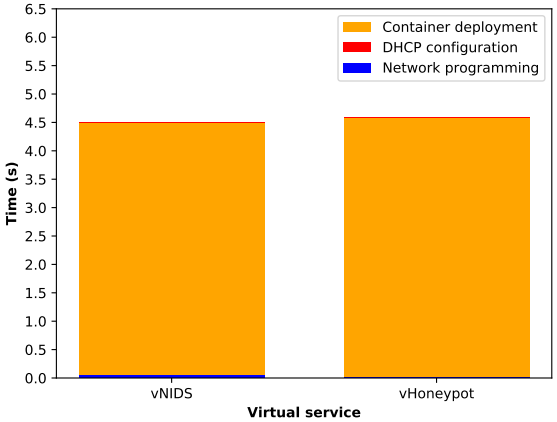


Figure 8.17: Time taken for each individual step in SDN probe deployment (3 host network).

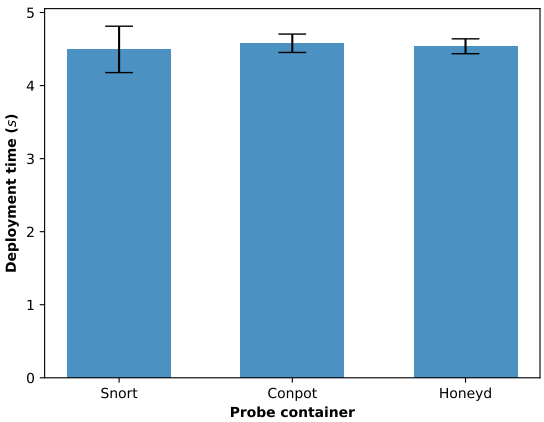


Figure 8.18: Probe container launch times depending on the container image.

Table 8.6: Elapsed time for each step involved in vProbe deployment.

vProbe	Step	Time (s)
vNIDS	Container launch	4.495 ± 0.317
	DHCP assign	0.000 ± 0.000
	Network prog.	0.051 ± 0.008
vHoneypot	Container launch	4.579 ± 0.126
	DHCP assign	0.003 ± 0.001
	Network prog.	0.016 ± 0.002

Table 8.7: Container deployment times depending on the vProbe container image

Probe	Container deployment time (s)
Snort	4.495 ± 0.317
Conpot	4.579 ± 0.126
Honeyd	4.538 ± 0.102

The performance results for the *vNIDS* application are shown in Figure 8.19 and summarized in Table 8.8. In this test, the container launch is not taken into account since an host is added to a previously created vNIDS service (with the respective container already started). It can be understood as the effect of the network and control cluster size on the network programming step of the vNIDS service.

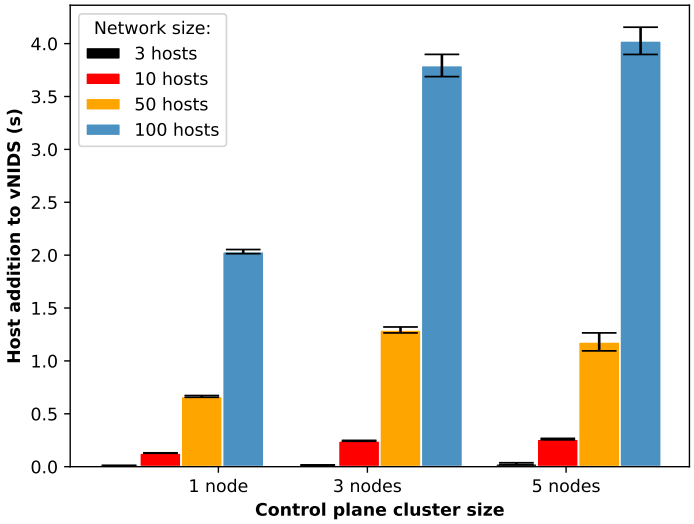


Figure 8.19: Host addition to a deployed vNIDS service depending on the overall network and control plane cluster sizes.

Through the analysis of the obtained results, it is possible to see the cluster size does not have a major influence in the network programming throughput of the *vNIDS application*. Contrarily to the *Network Management* application, the *vNIDS* application mainly modifies the previously installed (see Section 7.2). As a result, for a 100 host network, the addition of an host to the service can be achieved in just 3 to 4 seconds.

Table 8.8: Host addition to a deployed vNIDS service depending on the overall network and control plane cluster sizes.

Cluster size	Network size	vNIDS host addition latency (s)
1 node	3 hosts	0.011 ± 0.0
	10 hosts	0.129 ± 0.002
	50 hosts	0.664 ± 0.664
	100 hosts	2.034 ± 2.034
3 nodes	3 hosts	0.016 ± 0.001
	10 hosts	0.244 ± 0.004
	50 hosts	1.293 ± 0.028
	100 hosts	3.793 ± 0.105
5 nodes	3 hosts	0.028 ± 0.009
	10 hosts	0.261 ± 0.007
	50 hosts	1.18 ± 0.085
	100 hosts	4.026 ± 0.129

The same test was also executed for the *vHoneypot* application. Network programming in the *vHoneypot* application (c.f. Section 7.2) is simpler, with less flow rules, than in the *vNIDS* application. Hence, obtained times are smaller - the process is completed in 1 to 3 seconds.

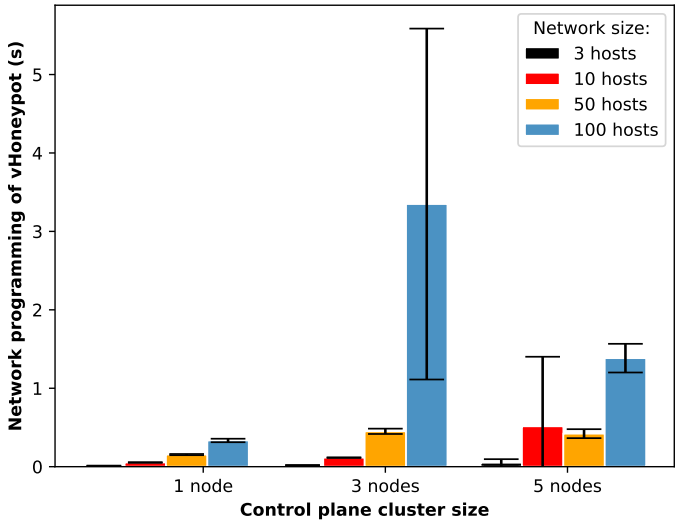


Figure 8.20: vHoneypot network programming (DHCP + flow rule installation).

It is also possible to note (as observed before in Section 8.3.1) a big variance value for the three node cluster when programming the vHoneypot for the 100 host network. This fact could indicate that a small control cluster may be unable to process such high amounts of flow-rule operations in short periods of time.

Table 8.9: vHoneypot network programming (DHCP + flow rule installation).

Cluster size	Network size	honeypot host addition latency (s)
1 node	3 hosts	0.012 ± 0.002
	10 hosts	0.053 ± 0.005
	50 hosts	0.154 ± 0.154
	100 hosts	0.334 ± 0.334
3 nodes	3 hosts	0.019 ± 0.002
	10 hosts	0.115 ± 0.004
	50 hosts	0.451 ± 0.034
	100 hosts	3.349 ± 2.237
5 nodes	3 hosts	0.05 ± 0.045
	10 hosts	0.515 ± 0.887
	50 hosts	0.421 ± 0.057
	100 hosts	1.384 ± 0.183

Data diode deployment

The data diode deployment test was similar to the ones outlined in the previous subsection. A CLI command was developed in the validation application to deploy and remove (25 times in a loop) a data diode on a known topology link. Results can be seen in Figure 8.21 and Table 8.10.

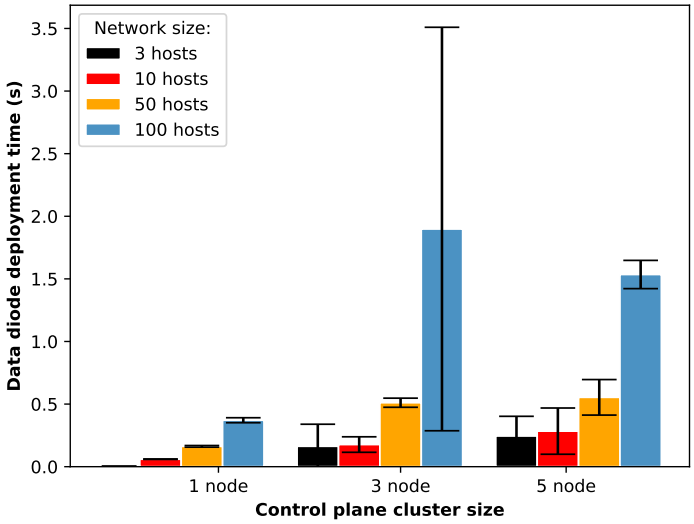


Figure 8.21: Data diode deployment times depending on the number of network hosts and control plane size.

The data diode is the network service achieving the fastest deployment times when compared to IADS vProbes. For small networks, the deployment of the data diode occurs in the millisecond range. For the biggest network of this test, the data diode deployment took only 1 to 2 seconds. A small value considering that for *n* hosts, *n-1* flow rules have to be installed and the datastore has to be consistently synchronized between all the controller nodes.

Table 8.10: Data diode deployment times depending on the number of network hosts and control plane size.

Cluster size	Network size	Data diode deployment time (s)
1 node	3 hosts	0.003 ± 0.0
	10 hosts	0.06 ± 0.002
	50 hosts	0.162 ± 0.162
	100 hosts	0.371 ± 0.371
3 nodes	3 hosts	0.163 ± 0.176
	10 hosts	0.177 ± 0.062
	50 hosts	0.511 ± 0.036
	100 hosts	1.898 ± 1.611
5 nodes	3 hosts	0.245 ± 0.157
	10 hosts	0.284 ± 0.185
	50 hosts	0.554 ± 0.142
	100 hosts	1.535 ± 0.113

Data plane performance

The overall SDN subsystem strongly depends on the capacity and link performance of the underlying data plane. The control plane and the respective developed applications may have a high throughput capacity when installing flow rules or reacting to network events but the OpenFlow switch fabric may represent a limiting effect on the supported bandwidth. Please

recall from Section 7.1 that DPDK was installed and configured in OpenvSwitch hosts to improve the performance of the data plane.

The performance evaluation was done using the *Iperf2* tool, by setting the Mininet machine as the server and the InventoryDB machine as the client. Both virtual machines are configured in passthrough mode in the hypervisor and have a dedicated physical gigabit interface. The theoretical maximum bandwidth in the link can then be assumed to be 1Gbps.

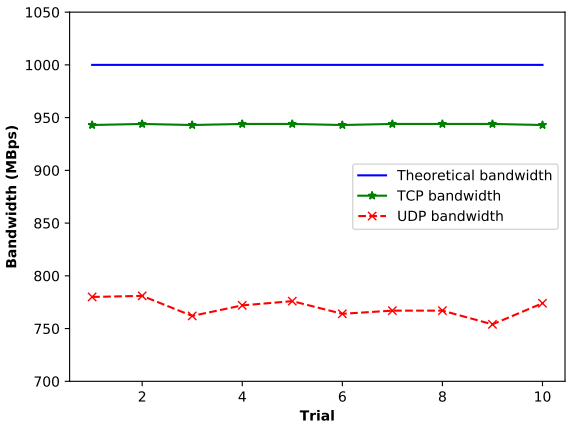


Figure 8.22: UDP bandwidth vs. TCP and Theoretical bandwidth.

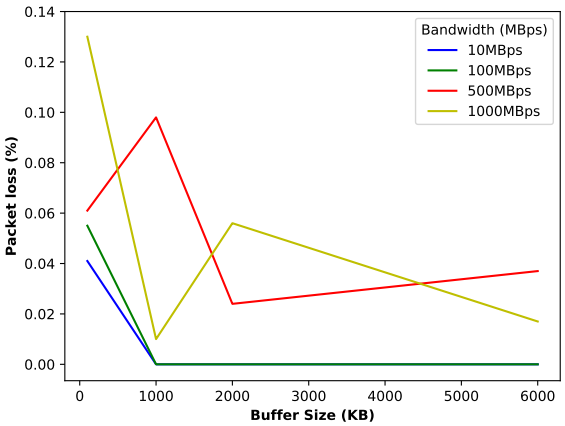


Figure 8.23: Percentage of lost packets vs. bandwidth and write buffer size (UDP).

The experiment was conducted in both TCP and UDP modes. The UDP test is important since in data diode links the traffic flow is unidirectional, so as the protocols used to carry data in the link. The TCP test achieved a value of 943.6 ± 0.36 Mbps, while the maximum bandwidth using UDP was 769.7 ± 7.4Mbps (cf. Figure 8.22)[1]. The TCP values for the maximum bandwidth were expected to be higher than the ones achieved by UDP since TCP automatically adjusts the window size during the transfer. Both values are comparable with some commercial switches, despite the software-based testbed.

Another test (Figure 8.23) was performed to study the effect of the sender packet buffer size on the obtained packet loss in UDP transfers. Please recall from subsection 8.2.3 that for a high number of sequential EMU readings, and due to the no-guarantee nature of the UDP protocol, some readings were not able to reach the RX machine. Hence, both the sender buffer size and the bandwidth were varied in this test. Measurements show that the buffer size plays a significant role on the packet loss, since it affects the total number of packets that can be sent in a single transfer. Thus, if the expected bandwidth is known beforehand (the case of SCADA traffic) both the sending and receiving agents can be optimized for minimal packet loss.

The last test regarding the data plane performance targeted the study of the latency in the path between the monitoring hosts and virtual probes. In the ATENA testbed, virtual probes (docker containers) are deployed in virtualization servers physically connected to the main testbed OpenFlow switch. Hence, for a packet to be copied to a vProbe, it has to traverse two switches: the main OpenFlow switch and the virtual switch deployed in the virtualization infrastructure (c.f Figure 8.1). A logical network was created between two hosts in the network, a vNIDS service was created in the network and both hosts were added

[1]Due to the low number of samples, bandwidth values were calculated using a t-student distribution for a 95% confidence level

to the vNIDS service. The *iptraf* tool (Paul 2005) was installed in the vProbe container to passively monitor the bandwidth reaching the container SDN interface.

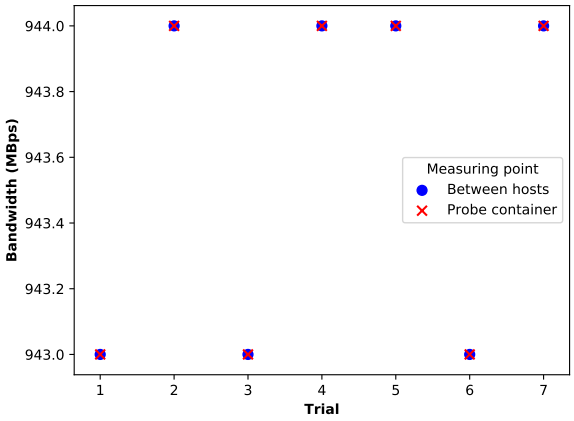


Figure 8.24: TCP bandwidth comparison: Host traffic *vs* vNIDS probe container.

Results for this test are plotted in Figure 8.24. The average bandwidth between the hosts being monitored and the peak bandwidth measured in the vProbe container denote an exact match. Thus, we can conclude the path to the container in the testbed (and the additional virtual switch) does not pose a significant effect on the performance of the monitoring process.

8.3.2 Availability

Availability tests served the purpose of quantifying the robustness of the created logical networks and the testbed performance in case of a controller node failure.

Intent fail-over

Logical networks in the *Network Management* application are created using the ONOS intent framework. This means the application basically tells the controller that connectivity is required between host A and host B without specifying the host location nor the respective flow rules.

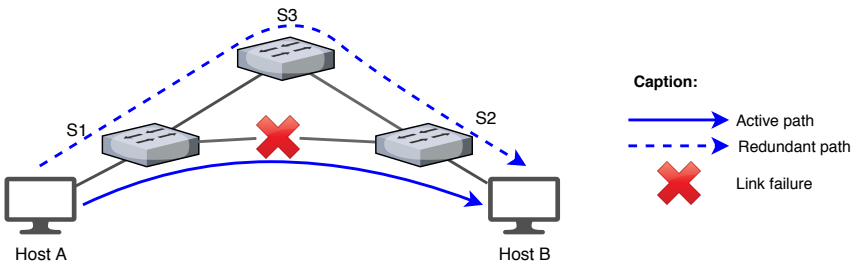


Figure 8.25: Link failure event and the selection of a redundant path.

ONOS internally keeps track of all installed intents and readjusts them in case of a link or switch failure as long as a redundant path exists between the hosts. A topology (represented in Figure 8.25) with two hosts and three switches (2 redundant paths) was generated in Mininet and a logical network was created between the two hosts.

By default, ONOS uses *Dijkstra's* algorithm to find the shortest path between hosts in the topology. So, when the logical network is first created the path between Host A and Host B uses S1 and S2. In the Mininet command line, a failure in the link between both switches was simulated:

```
> link s1 s2 down
```

The procedure was repeated 25 times while the validation application captured the sequence of generated *LINK_REMOVED* and *INTENT_INSTALLED* events. The latency (represented in Figure 8.26 and Table 8.11) is thus calculated upon the subtraction of both event timestamps. The SDN subsystem showed minimal latency values for intent readjustment operations. A redundant path is selected and the respective flow rules are installed in a period between 20 to 30 milliseconds.

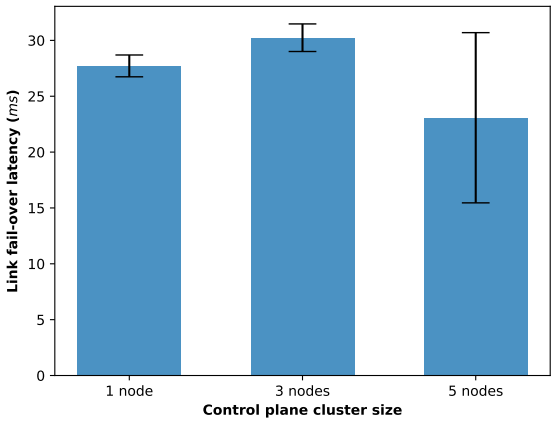


Figure 8.26: Intent fail-over latency depending on the control plane cluster size.

Table 8.11: Intent fail-over latency depending on the control plane cluster size.

Number of controller nodes	Intent fail-over latency (ms)
1	27.716 ± 0.977
3	30.239 ± 1.235
5	23.074 ± 7.619

The obtained results are extremely important since the overall fail-over process is much quicker than traditional network protocols used in the IACS field. For the Rapid Spanning Tree protocol (RSTP), the equivalent process usually takes a few seconds.

Switch mastership fail-over

In the switch mastership fail-over test, the elapsed time between a broken switch-to-controller connection and the respective mastership reassignment was measured. Using the implemented testbed, the main OpenFlow switch was associated with one specific controller node as master while keeping fallback connections to all the other controller nodes. The master node was rebooted while the validation application listened to the *INSTANCE_DEACTIVATED* and *MASTERSHIP_CHANGED* events. The latency was calculated as the mean value between both event timestamps.

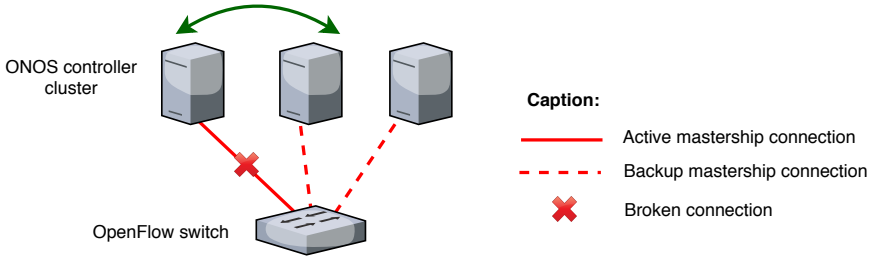


Figure 8.27: Switch mastership test representation.

Figure 8.27 shows a diagram representation of the test. Test results are presented in Figure 8.28 and Table 8.12.

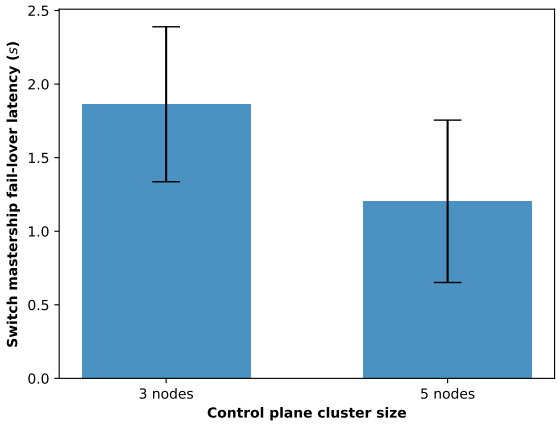


Figure 8.28: Switch mastership fail-over latency depending on the control plane cluster size.

Table 8.12: Switch mastership fail-over latency depending on the control plane cluster size.

Number of controller nodes	Switch mastership fail-over latency (s)
3	1.863 ± 0.527
5	1.203 ± 0.552

The obtained results show that an SDN controller cluster setup is able to reassign the mastership of a switch in approximately 1 to 2.5 seconds.

8.4 Chapter wrap-up

Globally, it is possible to conclude the subsystem proposed in this thesis meets its design goals: performance and availability. The functional validation scenarios showed the developed control plane applications could successfully take advantage of SDN and NFV to deploy containers and program the network according to the class of virtual service. The vNIDS container was able to passively obtain copies of selected network host traffic, while the vHoneypot container was able to behave as a generic host in the network. Data diode links, as expected, provided uni-directional communications while still keeping the rest of the logical network operational. Its validation also shown how bi-directional protocols can easily be ported to work in unidirectional links. The network event factory showed stability when operating for long periods of time, demonstrating how it can be an important point for event collecting (and statistics) in the overall IADS platform. The vNIDS functional validation was of extreme importance for the ATENA intermediate review success. The non-functional validation showed how network programming (flow rule installation) is a quick process in the IADS subsystem. Logical networks are created and virtual probes are deployed in a matter

of seconds - which is a market advantage if compared to the traditional, manual and slow process of traditional probe deployment. The data plane bandwidth tests demonstrated the work done to support DPDK in the testbed was beneficial: testbed switches had bandwidth values comparable to commercial switches, despite being software based. The non-functional validation also helped concluding that increasing the control plane size can both be an advantage (redundancy and performance) and a disadvantage (performance penalty due to synchronization). Its size should be carefully selected, depending on the overall goal and taking into account possible performance trade-offs. The proposed subsystem was able to operate even in the case of a link or controller node failure.

Chapter 9

Conclusions

Current trends, such as Industry 4.0 and Internet of Things, are evolving IACS towards ubiquity, moving away from the traditional monolithic and self-contained infrastructure paradigm, in favor of highly distributed and interconnected architectures. In this perspective, the management and monitoring of the critical infrastructure using traditional network architectures and traditional probe deployment may become extremely complex processes. Network equipment is provided by different vendors, relying on closed management protocols and different configuration instructions or interfaces. In the long term, it requires specialized training, it hampers innovation due to the closed nature of the protocols, and may lead to configuration errors. In the case of IACS, configuration mistakes create cyber-security holes which can lead to severe consequences. IACS support essential services such as the power grid and the water distribution systems; have cyber-physical implications and often depend on other critical infrastructures. Opting for turn-key solutions provided by a single enterprise (including technical support) is not really an alternative as it results in a complete vendor lock-in environment, imposing difficulties for innovation. Network passive monitoring is nowadays implemented in IACS with in-line physical probes (in the same path as other network devices), introducing latency and jitter in the network. As physical devices, they suffer from the problem of physical placement and can disrupt connectivity when flooded. This contradicts the IACS essential requirements: availability, performance and the need for real-time operations. Defining monitoring ports in a physical switch is also common. However, this approach lacks flexibility: (i) sometimes only a single port can be defined as a mirror port; (ii) switches often do not allow to select specific traffic patterns; and (iii) the process lacks a global network view and is still executed on a per-device basis. Software defined networks can help overcome the limitations enumerated above. By decoupling the data plane from the control plane, SDN can adopt open protocols and promote network programmability at a global level from a logical centralized location. Business value is shifted from the hardware level to crafted SDN applications with specific virtual network functions. SDN can also provide the means for new cyber-security techniques as it can effectively block traffic under certain circumstances and provide counters for packets traversing the network. Coupled with NFV, probes can deviate from the traditional physical model and move to the datacenter, consolidated in common hypervisors as virtual machines or containers.

SDN and NFV have been extensively evaluated in the literature as viable network alternatives for industrial control systems. However, the state-of-the-art review also found several points for possible improvement. Many of the proposed frameworks are merely conceptual, implemented in simple network controllers (often in Python) and with no source code available for direct reutilization. They do not explore the clustering nature of some SDN controllers and do not globally target performance. Single node controllers can represent a single point of failure in the architecture, violating the availability requirement of IACS. Many of the

studies implement external components which communicate with the controller via their REST interfaces to achieve flow rule installation. This results in high latency values for network operations that were supposed to occur in near real-time. With the rise of artificial intelligence (AI) and machine learning (ML), researchers are starting to evaluate SDN as a possible source for data mining and, ultimately, intrusion detection. Many research projects use a reactive approach, instructing OpenFlow switches to send every single packet header to the network controller for model training and validation. This process reduces the network bandwidth, creates latency in network flows, and can be exploited as denial-of-service attacks against the network controller. DDoS attacks are even more dangerous if SDN frameworks rely on single node network controllers. Another issue found in the reviewed literature was the lack of usability for the proposed tools: they do not contemplate multi-tenancy nor role segregation (essential in IACS). Furthermore, they do not take advantage of the controller web interfaces and create steep learning curves in a field where innovation moves slowly.

The ATENA project, and more specifically, the IADS framework developed by the University of Coimbra, proposed to solve many of the enumerated issues by creating an highly decentralized and distributed platform for intrusion detection - relying on machine learning (and big data) techniques. The SDN subsystem of IADS proposed in this thesis can leverage this infrastructure to properly collect any meaningful events and delegate further processing to the upper layers of the IADS architecture. In the SDN subsystem, network functions were implemented as abstracted services tied to logical sections of the network (the root basis for multi-tenancy) and flow rules are installed using a proactive approach. The logic is implemented directly at the application layer, using the controller native APIs, and is exposed via external interfaces only for visualization purposes. A distributed (clustered) controller was used to support the system and carefully select according to performance metrics available in the literature. The subsystem includes a user-friendly web interface, with different privileged levels, where service deployment was achieved in the network topology graph. This helped reducing the learning curve and the likelihood of SDN adoption as the model is similar to HMIs already being used in IACS for control operations. The proposed subsystem also explores container-based virtualization, combining it with the OpenFlow protocol for container networking – a novel approach not yet explored in the IACS. It took advantage of the docker container engine and its built-in image management capabilities to store probe images/templates on a private registry. The registry separates images according to the type of service, and allows the administrator to upload and remove new images to the system – creating an environment similar to a security probe store. The system also allows scaling the number of virtualization servers available for probe deployment. A total of four services were developed and explored for the IACS domain:

- vNIDS - a virtual intrusion detection system,
- vHoneypot - a virtual honeypot service, designed for IACS,
- Data diode - a virtual SDN enabled data diode service,
- Network event factory - a special type of probe that piped SDN events to the upper layers of the platform,

matching the goals outlined for this implementation component of this thesis. Probe deployment in IADS happens in a matter of seconds (mostly due to container launching) and the network programming step can be neglected in the overall logical network is small. The validation of the subsystem allowed to conclude that every service behave exactly as expected considering the security use-cases outlined in the test methodology. Non-functional

validation confirmed the main design quality attributes: performance and availability. The system is able to operate even in cases of a link, switch or controller node failure showing there is not a complete dependency on the control plane. The validation also permitted to conclude that scaling the control plane size can represent both an advantage or a disadvantage in terms of performance. Its size should be carefully selected depending on the situation. Regarding the data plane, the efforts to improve performance through DPDK installation shown OpenFlow software switches can have similar bandwidth values if compared to commercial switches. The path to the virtualization infrastructure does not introduce significant latency in the monitoring process as the same bandwidth was obtained within virtual probe containers. Overall, the virtualized services developed in this thesis can thus compare favorably with traditional approaches while maintaining functional equivalence to their physical counterpart.

9.1 Suggestions for future work

The developed system is not yet able to automatically scale the number of deployed containers for a given service as the network traffic reaching a vProbe increases. Developed applications provide the necessary APIs to achieve it but monitor components need to be developed to periodically check the deployed container's state and reacting accordingly. Scale-up/scale-down mechanisms in IADS are, at the moment, a manual process. If scalability policy support is added to the subsystem, value would be added as its model would be more close to the one provided by the *cloud computing* paradigm. New containers would be launched automatically, with changed sets of flow rules, to reduce the load of existing containers.

A worth note to the future of *Software defined networking* is also imperative. The SDN architecture is a relatively new field and new advances have been proposed in recent years. Contrarily to Computer Networking domain itself, SDN is an area were innovation moves relatively fast. Originated in 2008, the OpenFlow protocol is starting to get deprecated in favor of new approaches such as the P4 programming language. In fact, despite allowing data plane programming from the control plane, OpenFlow includes a lot of features that, depending on the use case, are often not needed. In this research work, action buckets, meters or QoS mechanisms are concrete examples of features that are built-in into each switch and are not explored. P4 uses a different approach, it represents an abstraction layer that sits above the vendor firmware. It can be used to program the switch behavior (flow rules), delegate decisions to a network controller or add support for newer protocols in the switch. P4 is not exactly a replacement for OpenFlow but a complementary technology to add support for a multitude of protocols (including OpenFlow) on the same hardware. Like OpenFlow, switches are treated as "open"/"empty" boxes and it is up to the programmer to define the range of protocols or implicit data plane behavior it should have. In fact, ONOS is moving in that direction too: the big majority of the recent developments are focused on P4 support rather than updating the OpenFlow support for higher protocol revisions. In the IADS subsystem P4 could be used to clearly define what rules are allowed in each switch at installation time, similarly to a "firmware update", and to limit the code base (and inherently the network switch attack surface) to a minimum.

Bibliography

Abubakar, Atiku et al. (2017). "Machine Learning Based Intrusion Detection System for Software Defined Networks". In: *Proceedings of the 2017 Eighth International Conference on Emerging Security Technologies (EST). IEEE.*

Ajaeiya, Georgi A. et al. (2017). "Flow-based Intrusion Detection System for SDN". In: *Proceedings - IEEE Symposium on Computers and Communications*, pp. 787–793. isbn: 9781538616291. doi: 10.1109/ISCC.2017.8024623.

Alcaraz, Cristina et al. (2015). "Security Aspects of SCADA and DCS Environments". In: pp. 1–32.

Argyropoulos, C. et al. (2015). "Control-plane slicing methods in multi-tenant software defined networks". In: *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015*, pp. 612–618. isbn: 9783901882760. doi: 10.1109/INM.2015.7140345. arXiv: arXiv:1307.8198v1.

ATENA (2017). *ATENA project official website*. [Online] Accessed 3 January 2018. url: https://www.atena-h2020.eu/.

ATENA youtube (2018). *IADS demo 2 videos*. [Online] Accessed 21 January 2018. url: https://www.youtube.com/watch?v=NxLZ2Sqtujk%7B%5C&%7Dlist=PLyl2zhykZrH8nHidjvjG5B-9OKOw3uDd-.

AtenaConsortium4.1 (2017). *D4.1: Requirements and Reference Architecture for the Cyber-physical IDS*. Project Deliverable. UC, UL, IEC, UNIROMA3, iTRUST, ENEA.

AtenaConsortium4.3 (2017). *D4.3 – Design of Detection Agents and Security Components*. Project Deliverable. UC, UL, iTrust.

Automation.com (2017). *Yokogawa enhances plant network for four Japanese paper plants*. Accessed: 2017-12-20.

Axios (2018). *Axios library*. [Online] Accessed 21 January 2018. url: https://github.com/axios/axios.

Aydeger et al. (2016). "SDN-based resilience for smart grid communications". In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network, NFV-SDN 2015*, pp. 31–33. isbn: 9781467368841. doi: 10.1109/NFV-SDN.2015.7387401.

Azevedo F. (2015). "A Scalable Architecture for OpenFlow SDN Controllers". MA thesis. Lisboa, Portugal: IST-UL.

Azodolmolky, Siamak (2013). *Software Defined Networking with OpenFlow*. Packt Publishing. isbn: 1849698724, 9781849698726.

Banse, Christian et al. (2015). "A secure northbound interface for SDN applications". In: *Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015*. Vol. 1, pp. 834–839. isbn: 9781467379519. doi: 10.1109/Trustcom.2015.454.

Barbosa, R.R.R. (2014). *Anomaly detection in SCADA systems: a network based approach*. doi: 10.3990/1.9789036536455.

Berde, Pankaj et al. (2014). "ONOS: towards an open, distributed SDN OS". In: *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, pp. 1–

6. doi: 10.1145/2620728.2620744. url: http://dl.acm.org/citation.cfm?id=2620728.2620744.

B.Freitas, M (2017). *Allow multiple VNC windows*. url: https://github.com/OpenXenManager/openxenmanager/pull/132.

– (2018). *Remove host location when a switch port is removed*. url: https://gerrit.onosproject.org/#/c/18996/.

Biondi, Philippe (2018). *Scapy - Packet crafting for Python2 and Python3*. [Online] Accessed 31 August 2018. url: https://scapy.net/.

Brewer, Ea (2000). "Towards Robust Distributed Systems". In: *Podc*, p. 50. issn: 01635700. doi: 10.1145/343477.343502.

Brikman, Y. (2017). *Terraform: Up and Running : Writing Infrastructure as Code*. O'Reilly Media. isbn: 9781491977088. url: https://books.google.pt/books?id=MLkRMQAACAAJ.

Cadenas, Manuel et al. (2016). *BTest - A performance analysis tool for SDN controllers: OpenDayLight versus ONOS comparison*. http://onos-cord-eu.create-net.org/wp-content/uploads/2016/09/07-Btest-RA-1.pdf.

Cagalaban, Giovanni et al. (2011). "Towards Improving SCADA Control Systems Security with Vulnerability Analysis". In: *PARALLEL AND DISTRIBUTED COMPUTING AND NETWORKS*. Vol. 137, pp. 27–32. isbn: 978-3-642-22705-9.

Cahn, Adam et al. (2013). "Software-defined energy communication networks: From substation automation to future smart grids". In: *2013 IEEE International Conference on Smart Grid Communications, SmartGridComm 2013*, pp. 558–563. isbn: 9781479915262. doi: 10.1109/SmartGridComm.2013.6688017.

Cheung, Steven et al. (2006). "Using Model-based Intrusion Detection for SCADA Networks". In: *Science And Technology* 329.7461, pp. 1–12. issn: 09598138. doi: 10.1136/bmj.329.7461.331.

Cisco (2013). *OpenDaylight: The Start of Something Big for SDN*. http://blogs.cisco.com/datacenter/opendaylight-the-start-of-something-big-for-sdn. [Online] Accessed 12 June 2017.

Cockburn, Alistair (2000). *Writing Effective Use Cases*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. isbn: 0201702258.

Combe, Theo et al. (2016). "To Docker or Not to Docker: A Security Perspective". In: *IEEE Cloud Computing* 3.5, pp. 54–62. issn: 23256095. doi: 10.1109/MCC.2016.100.

Cox, Jacob H. et al. (2017). "Advancing Software-Defined Networks: A Survey". In: *IEEE Access* 99, pp. 1–1. issn: 2169-3536. doi: 10.1109/ACCESS.2017.2762291. url: http://ieeexplore.ieee.org/document/8066287/.

Cruz, Tiago et al. (2016). "Virtualizing programmable logic controllers: Toward a convergent approach". In: *IEEE Embedded Systems Letters* 8.4, pp. 69–72. issn: 19430663. doi: 10.1109/LES.2016.2608418.

Cruz, Tiago et al. (2015). "Improving network security monitoring for industrial control systems". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pp. 878–881. isbn: 978-1-4799-8241-7. doi: 10.1109/INM.2015.7140399. url: http://ieeexplore.ieee.org/document/7140399/%7B%5C%%7D5Cnhttp://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7140399.

Cziva, Richard et al. (2016). "Container-based network function virtualization for software-defined networks". In: *Proceedings - IEEE Symposium on Computers and Communications*. Vol. 2016-February, pp. 415–420. isbn: 9781467371940. doi: 10.1109/ISCC.2015.7405550.

Da Silva et al. (2015). "Capitalizing on SDN-based SCADA systems: An anti-eavesdropping case-study". In: *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015*, pp. 165–173. isbn: 9783901882760. doi: 10.1109/INM.2015.7140289.

Da Silva, Eduardo Germano et al. (2016). "A One-Class NIDS for SDN-Based SCADA Systems". In: *Proceedings - International Computer Software and Applications Conference*. Vol. 1, pp. 303–312. isbn: 9781467388450. doi: 10.1109/COMPSAC.2016.32.

Darianian, Mohamad (2017). "Experimental Evaluation of Two OpenFlow Controllers". MA thesis. Calgary, Alberta: University of Calgary.

De Freitas, Breno Jácomo (2012). "Preventive actions in protection relays network using SNMP". In: *2012 11th International Conference on Environment and Electrical Engineering, EEEIC 2012 - Conference Proceedings*, pp. 36–40. isbn: 9781457718281. doi: 10.1109/EEEIC.2012.6221401.

Dharani, R. (2017). *Web API Design: Crafting Interfaces That Developers Love*. Independently Published. isbn: 9781973436249. url: https://books.google.pt/books?id=ow8ZuAEACAAJ.

Docker Documentation (2017). *About images, containers, and storage drivers*. [Online] Accessed 11 December 2017. url: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/.

Docker Networking (2017). *Docker container networking*. [Online] Accessed 7 December 2017. url: https://docs.docker.com/engine/userguide/networking/#user-defined-networks.

Dong, Xinshu et al. (2015). "Software-Defined Networking for Smart Grid Resilience: Opportunities and Challenges". In: *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, pp. 61–68. issn: 21615330. doi: 10.1145/2732198.2732203. url: http://doi.acm.org/10.1145/2732198.2732203.

Drias, Zakarya et al. (2015). "Taxonomy of attacks on industrial control protocols". In: *International Conference on Protocol Engineering, ICPE 2015 and International Conference on New Technologies of Distributed Systems, NTDS 2015 - Proceedings*. isbn: 9781467392655. doi: 10.1109/NOTERE.2015.7293513. arXiv: 0-387-31073-8.

Ecmweb (2003). *The Basics of Ladder Logic*. http://www.ecmweb.com/archive/basics-ladder-logic. [Online] Accessed 13 January 2018.

Eder, Michael et al. (2016). "Hypervisor- vs. Container-based Virtualization". In: *Network Architectures and Services* July, pp. 1–7. doi: 10.2313/NET-2016-07-1.

Endi, Mohamed et al. (2010). "Three-Layer PLC / SCADA System Architecture in Process Automation and Data Monitoring". In: pp. 774–779.

Eric Byres (2016). *The Industrial Cybersecurity Problem - ISA*. [Online] Accessed 16 December 2017. url: https://www.isa.org/pdfs/the-industrial-cybersecurity-problem/.

eTutorials.org (2008). *Quality Attribute Scenarios in Practice*. http://etutorials.org/Programming/Software+architecture+in+practice,+second+edition/Part+Two+Creating+an+Architecture/Chapter+4.+Understanding+Quality+Attributes/4.4+Quality+Attribute+Scenarios+in+Practice/. Accessed 5 July 2017.

Feamster, Nick et al. (2014). "The road to SDN". In: *ACM SIGCOMM Computer Communication Review* 44.2, pp. 87–98. issn: 01464833. doi: 10.1145/2602204.2602219. url: http://dl.acm.org/citation.cfm?doid=2602204.2602219.

Fernandez, Marcial P. (2013). "Comparing OpenFlow controller paradigms scalability: Reactive and proactive". In: *Proceedings - International Conference on Advanced Information*

Networking and Applications, AINA, pp. 1009–1016. isbn: 9780769549538. doi: 10.1109/AINA.2013.113.

Filipova, O. (2016). *Learning Vue.js 2*. Packt Publishing, Limited. isbn: 9781786469946. url: https://books.google.pt/books?id=q0FkvgAACAAJ.

Fireeye (2017). *Attackers Deploy New ICS Attack Framework "TRITON" and Cause Operational Disruption to Critical Infrastructure*. [Online] Accessed 18 January 2018. url: https://www.fireeye.com/blog/threat-research/2017/12/attackers-deploy-new-ics-attack-framework-triton.html.

Forbes, Harry (2017). "Software-defined Industrial Networks Deliver Cybersecurity Breakthroughs". In: *ARC BRIEF*.

FortFox (2010). *Security Target Common Criteria FFHDD – EAL7+*. Tech. rep. FortFox. url: https://www.commoncriteriaportal.org/files/epfiles/Fox%20DataDiode%20Security%20Target%20EAL7%20(v2.04).pdf.

Fotrousi, Farnaz et al. (2014). "Quality requirements elicitation based on inquiry of quality-impact relationships". In: *2014 IEEE 22nd International Requirements Engineering Conference, RE 2014 - Proceedings*, pp. 303–312. isbn: 9781479930333. doi: 10.1109/RE.2014.6912272.

Fysarakis, Konstantinos et al. (2017). "A Reactive Security Framework for operational wind parks using Service Function Chaining". In: *Proceedings - IEEE Symposium on Computers and Communications*, pp. 663–668. isbn: 9781538616291. doi: 10.1109/ISCC.2017.8024604.

Ghosh, Uttam et al. (2017). "A Security Framework for SDN-Enabled Smart Power Grids". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 113–118. doi: 10.1109/ICDCSW.2017.20. url: http://ieeexplore.ieee.org/document/7979803/.

Ghosh, Uttam et al. (2016). "A Simulation Study on Smart Grid Resilience under Software-Defined Networking Controller Failures". In: *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security - CPSS '16*, pp. 52–58. isbn: 9781450342889. doi: 10.1145/2899015.2899020. url: http://dl.acm.org/citation.cfm?doid=2899015.2899020.

Giacobbi, Giovanni (2006). *The GNU Netcat project*. [Online] Accessed 31 August 2018. url: http://netcat.sourceforge.net/.

Goodney, Andrew et al. (2013). "Efficient PMU networking with software defined networks". In: *2013 IEEE International Conference on Smart Grid Communications, SmartGridComm 2013*, pp. 378–383. isbn: 9781479915262. doi: 10.1109/SmartGridComm.2013.6687987.

Goransson, Paul et al. (2014). *Software Defined Networks: A Comprehensive Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. isbn: 012416675X, 9780124166752.

Graveto, Vitor (2017). "Evolving Monitoring Approaches for Cyber-Physical Systems". Ph.D. Research Proposal. University of Coimbra.

Gyorgy, Kalman (2016). "Prospects of Software-Defined Networking in Industrial Operations". In: 9.3, pp. 101–110.

Haack, P. (2011). *Release Early, Release Often*. http://haacked.com/archive/2011/04/20/release-early-and-often.aspx. [Online] Accessed July 2017.

Han, Wonkyu et al. (2016). "HoneyMix: Toward SDN-based Intelligent Honeynet". In: *... in Software Defined Networks & ...* Pp. 1–6. doi: 10.1145/2876019.2876022. url: http://dl.acm.org/citation.cfm?id=2876022.

Heo, Youngjun et al. (2016). "A design of unidirectional security gateway for enforcement reliability and security of transmission data in industrial control systems". In: *International Conference on Advanced Communication Technology, ICACT*. Vol. 2016-March, pp. 310–313. isbn: 9788996865063. doi: 10.1109/ICACT.2016.7423372.

Hernandez, Esteban (2016). "Implementation and Performance of a SDN Cluster Controller Based on the OpenDayLight Framework". MA thesis. Milano: Politecnico di Milano.

hicu.be (2016). *Macvlan vs Ipvlan*. [Online] Accessed 7 December 2017. url: https://hicu.be/macvlan-vs-ipvlan.

Huang, Nen Fu et al. (2015). "An OpenFlow-based collaborative intrusion prevention system for cloud networking". In: *Proceedings of 2015 IEEE International Conference on Communication Software and Networks, ICCSN 2015*, pp. 85–92. isbn: 9781479919833. doi: 10.1109/ICCSN.2015.7296133.

Hughey, D. (2017). *Comparing Traditional Systems Analysis and Design with Agile Methodologies*. http://www.umsl.edu/ hugheyd/is6840/waterfall.html. Accessed July 2017.

IBM Corp (2006). *Guideline: Use-Case Package*. http://www.michael-richardson.com/processes/rup_for_sqa/core.base_rup/guidances/guidelines/use-case_package_1EFD6458.html. Accessed 8 July 2017.

IEEE (1994). *IEEE recommended practice for software requirements specifications*. IEEE Std. Institute of Electrical and Electronics Engineers. isbn: 9781559373951. url: https://books.google.pt/books?id=CnopAQAAMAAJ.

IEEE Std 1233 (1998). *IEEE Std 1233, 1998 Edition: IEEE Guide for Developing System Requirements Specifications*. IEEE. url: https://books.google.pt/books?id=f70%5C_nQAACAAJ.

InfoWorld (2017). *What is Docker? Linux containers explained*. [Online] Accessed 15 December 2017. url: https://www.infoworld.com/article/3204171/linux/what-is-docker-linux-containers-explained.html.

ISA99 (2017). *ISA99: Developing the ISA/IEC 62443 Series of Standards on Industrial Automation and Control Systems (IACS)*. [Online] Accessed 28 December 2017. url: http://isa99.isa.org/.

ISO/IEC (2010). *ISO/IEC 25010 System and software quality models*. Tech. rep.

J. Goss (2007). *10 reasons why use cases are indispensable to your software development project*. http://www.techrepublic.com/blog/software-engineer/10-reasons-why-use-cases-are-indispensable-to-your-software-development-project. Accessed 19 July 2017.

Jeong, Chiwook et al. (2014). "Scalable network intrusion detection on virtual SDN environment". In: *2014 IEEE 3rd International Conference on Cloud Networking, CloudNet 2014*, pp. 264–265. isbn: 9781479927302. doi: 10.1109/CloudNet.2014.6969003.

K. M. Anderson (2005). *Lecture 7 and 8: Use Cases CS-Colorado*. https://www.cs.colorado.edu/~kena/classes/6448/s05/lectures/lecture07-08.pdf. Accessed 6 July 2017.

Kaspersky (2016). *Threat landscape for industrial automation systems in the second half of 2016*. [Online] Accessed 16 January 2017. url: https://ics-cert.kaspersky.com/reports/2017/03/28/threat-landscape-for-industrial-automation-systems-in-the-second-half-of-2016/.

Khattak, Zuhran et al. (2014). "Performance evaluation of OpenDaylight SDN controller". In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*. Vol. 2015-April, pp. 671–676. isbn: 9781479976157. doi: 10.1109/PADSW.2014.7097868.

Kim Zetter (2015). *Inside the cunning, unprecedented hack of Ukraine's power grid*. [Online] Accessed 16 January 2018. url: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/.

Kim, Jaebeom et al. (2015). "Trends and potentials of the smart grid infrastructure: from ICT sub-system to SDN-enabled smart grid architecture". In: *Applied Sciences* 5.4, pp. 706–727.

Kreutz, Diego et al. (2014). "Software-Defined Networking: A Comprehensive Survey". In: *arXiv preprint arXiv: ...* P. 49. issn: 0018-9219. doi: 10.1109/JPROC.2014.2371999. arXiv: 1406.0440. url: http://arxiv.org/abs/1406.0440.

Kurtz, Fabian et al. (2017). "Advanced Controller Resiliency in Software-Defined Networking Enabled Critical Infrastructure Communications". In: pp. 673–678.

Lallo, Roberto di et al. (2017). "Leveraging SDN to monitor critical infrastructure networks in a smarter way". In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 608–611. doi: 10.23919/INM.2017.7987341. arXiv: 1701.04293. url: http://ieeexplore.ieee.org/document/7987341/.

Lamport, Leslie et al. (2014). "In Search of an Understandable Consensus Algorithm". In: *Atc '14* 22.2, pp. 305–320. issn: 07342071. doi: 10.1145/1529974.1529978. arXiv: 1505.01448.

Lara, Adrian et al. (2013). "Network Innovation using OpenFlow: A Survey". In: *IEEE Communications Surveys & Tutorials* PP.99, pp. 1–20. issn: 1553-877X. doi: 10.1109/SURV.2013.081313.00105.

Li, Yong et al. (2015). "Software-defined network function virtualization: A survey". In: *IEEE Access* 3, pp. 2542–2553. issn: 21693536. doi: 10.1109/ACCESS.2015.2499271.

LinuxFoundation (2014). *Developing OpenDaylight Apps with MD-SAL*. https://events.static.linuxfound.org/sites/events/files/slides/os2014-md-sal-tutorial_0.pdf. Accessed 11 June 2017.

Lopes et al. (2017). "A Security Framework for SDN-Enabled Smart Power Grids". In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. doi: 10.23919/INM.2017.7987283.

MachineryEquipment (2015). *Industrial control networks: supervisory control and data acquisition (SCADA) network*. [Online] Accessed 15 January 2018. url: http://machineryequipmentonline.com/electric-equipment/industrial-control-networkssupervisory-control-and-data-acquisition-scada-network/.

Manzano, Andrés et al. (2016). "A prototype for a honeynet based on SDN". In: *2016 8th Euro American Conference on Telematics and Information Systems, EATIS 2016*. isbn: 9781509024360. doi: 10.1109/EATIS.2016.7520100.

MapDB (2018). *MapDB about page*. http://www.mapdb.org/. Accessed 4 January 2018.

Marinescu, Dan et al. (2007). *State of the art in autonomic computing and virtualization*. SoA. Distributed Systems Lab, Wiesbaden University of Applied Sciences.

Mckay, Murray (2012). "Best practices in automation security". In: url: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1001.6096%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

Menashri, Harel et al. (2015). "Critical Infrastructures and their Interdependence in a Cyber Attack – The Case of the U.S". In: *Military and Strategic Affairs* 7.1, p. 22. url: http://www.inss.org.il/uploadImages/systemFiles/5%7B%5C_%7DMenashri%7B%5C_%7DBaram.pdf.

Microsoft (2017). *Chapter 16: Quality Attributes, from Microsoft, Developer Network*. https://msdn.microsoft.com/en-us/library/ee658094.aspx. Accessed 4 July 2017.

Minicz et al. (2017). "Fault Recovery Performance in Multicast Networks for Smart Grid". In: *IEEE Latin America Transactions* 15.11, pp. 2207–2213. issn: 15480992. doi: 10.1109/TLA.2017.8070428.

Mirantis (2015). *What's in OpenDaylight?* https://www.mirantis.com/blog/whats-opendaylight/. Accessed 10 June 2017.

Mo, Yilin et al. (2012). "Cyber-physical security of a smart grid infrastructure". In: *Proceedings of the IEEE* 100.1, pp. 195–209. issn: 00189219. doi: 10.1109/JPROC.2011.2161428.

Molina, Elias et al. (2015). "Using Software Defined Networking to manage and control IEC 61850-based systems". In: *Computers and Electrical Engineering* 43, pp. 142–154. issn: 00457906. doi: 10.1016/j.compeleceng.2014.10.016.

Monshizadeh et al. (2017). "Detection as a service: An SDN application". In: *International Conference on Advanced Communication Technology, ICACT*, pp. 285–290. issn: 17389445. doi: 10.23919/ICACT.2017.7890099.

Moradi, Farnaz et al. (2017). "ConMon: An automated container based network performance monitoring system". In: *Proceedings of the IM 2017 - 2017 IFIP/IEEE International Symposium on Integrated Network and Service Management*, pp. 54–62. doi: 10.23919/INM.2017.7987264.

Mousa, Mohammad et al. (2016). "Software Defined Networking concepts and challenges". In: *2016 11th International Conference on Computer Engineering & Systems (ICCES)*, pp. 79–90. doi: 10.1109/ICCES.2016.7821979. url: http://ieeexplore.ieee.org/document/7821979/.

MSec (2017). *Industrial Control Systems ARCHITECTURES & SECURITY ESSENTIALS*. https://www.msec.be/verboten/seminaries/ICS_archs_and_sec_essentials/ICS_Overview.pdf. [Online] Accessed: 2018-01-2.

Nagesh, Osri et al. (2017). "A survey on security aspects of server virtualization in cloud computing". In: *International Journal of Electrical and Computer Engineering* 7.3, pp. 1326–1336. issn: 20888708. doi: 10.11591/ijece.v7i3.pp1326-1336.

Naseer, Muhammad (2016). "Modeling Control Traffic in Distributed Software Defined Networks". MA thesis. Stockholm, Sweden: KTH ROYAL INSTITUTE OF TECHNOLOGY SCHOOL OF ELECTRICAL ENGINEERING.

NetworkStatic (2013). *OpenFlow: Proactive vs Reactive Flows*. [Online] Accessed 27 December 2017. url: http://networkstatic.net/openflow-proactive-vs-reactive-flows/.

Neu, Charles V. et al. (2017). "An approach for detecting encrypted insider attacks on OpenFlow SDN Networks". In: *2016 11th International Conference for Internet Technology and Secured Transactions, ICITST 2016*, pp. 210–215. isbn: 9781908320735. doi: 10.1109/ICITST.2016.7856698.

Norman, D. (2013). *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books. isbn: 9780465072996. url: https://books.google.pt/books?id=I1o4DgAAQBAJ.

ODL-license (2017). *OpenDaylight Licensing*. https://www.opendaylight.org/licensing. Accessed 12 June 2017.

ODL-team (2017). *OpenFlow Plugin Project Developer Guide*. http://docs.opendaylight.org/en/stable-boron/developer-guide/openflow-plugin-project-developer-guide.html. Accessed 13 January 2018.

ODLWiki-clustering (2017). *OpenDaylight/ Controller:MD-SAL:Architecture:Clustering*. https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Architecture:Clustering. Accessed 18 January 2018.

O.N.F. (2012). "Software-defined networking: The new norm for networks". In: *ONF White Paper* 2, pp. 2–6.

ONF-ESTI (2015). *Relationship of SDN and NFV*. https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/onf2015.310_Architectural_comparison.08-2.pdf. Accessed 15 January 2018.

On.Lab (2017). *Raising the bar on SDN performance, scalability, and high availability*. Tech. rep. On.Lab. url: http://onosproject.org/wp-content/uploads/2017/08/ONOS_Performance_White_Paper-2.pdf.

ONOS testcases (2018). *Experiment C - Intent Install/Remove/Re-route Latency*. [Online] Accessed 21 January 2018. url: https://wiki.onosproject.org/pages/viewpage.action?pageId=23332278.

ONOS-JAVAdocs (2018). *ONOS JAVA docs*. http://api.onosproject.org/1.12.0/. Accessed 4 January 2018.

ONOSproject (2016). *ONOS blog - ON.Lab and The Linux Foundation Form CORD Project to Define the Future of Access*. https://onosproject.org/2016/07/26/on-lab-and-the-linux-foundation-form-cord-project-to-define-the-future-of-access/. Accessed 4 January 2018.

ONOS-team (2016). *OpenDaylight & ONOS Performance white-paper*. https://onosproject.org/2016/05/20/opendaylight-onos-performance-white-paper/.

ONOS-website (2018). *ONOS project members*. https://onosproject.org/members/. [Online] Accessed 4 January 2018.

ONOS-wiki1 (2018). *ONOS Distributed Primitives*. https://wiki.onosproject.org/display/ONOS/Distributed+Primitives. Accessed 4 January 2018.

ONOS-wiki2 (2018). *ONOS Cluster Coordination*. https://wiki.onosproject.org/display/ONOS/Cluster+Coordination. Accessed 4 January 2018.

ONOS-wiki3 (2018). *ONOS System Components*. https://wiki.onosproject.org/display/ONOS/System+Components. Accessed 4 January 2018.

ONOS-wiki4 (2018). *Troubleshooting ONOS OSGi components*. https://wiki.onosproject.org/display/ONOS/Troubleshooting+ONOS+OSGi+components. Accessed 4 January 2018.

ONOS-wiki5 (2018). *ONOS intent framework*. https://wiki.onosproject.org/display/ONOS/Intent+Framework. Accessed 4 January 2018.

Open NF (2015). *OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06)*. [Online] Accessed 20 December 2017. url: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

OpenCORD (2018). *OpenCORD website*. https://opencord.org/. Accessed 4 January 2018.

OpenDayLight (2018). *OpenDaylight Boron: Platform For Network-Driven Businesses*. https://www.opendaylight.org/what-we-do/current-release/boron. Accessed 18 January 2018.

OpenDayLight-Tutorial (2016). *OpenDayLight: installing Berillium*. https://blog.rojerfarre.com/2016/03/01/opendaylight-installing-beryllium/. Accessed 15 June 2017.

OpenDayLight-Wiki (2018). *Messaging4Transport:AMQP Bindings for MD-SAL*. https://wiki.opendaylight.org/view/Messaging4Transport:AMQP_Bindings_for_MD-SAL. Accessed 18 January 2018.

OPNFV (2018). *OPNFV Euphrates documentation*. http://docs.opnfv.org/en/stable-euphrates/index.html. [Online] Accessed 15 January 2018.

Oracle (2018a). *JAX-RS documentation*. [Online] Accessed 20 August 2018. url: https://docs.oracle.com/javaee/7/api/javax/ws/rs/package-summary.html.

– (2018b). *Jersey framework*. [Online] Accessed 20 August 2018. url: https://jersey.github.io/.

OSGiAlliance (2018). *OSGi Alliance wiki*. https://www.osgi.org/community/wiki/. Accessed 4 January 2018.

OVS-Docker Github (2017). Accessed: 2017-12-7.

OWASP-Project (2017). *OWASP Project Home Page*. https://www.owasp.org/. [Online] Accessed 12 June 2017.

OWASP-Zap (2018). *OWASP Zap home page*. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. Accessed 15 January 2018.

Paliou, Despoina (2016). "SDN Application for Dynamic Routing in Software Defined Networks". MA thesis. Athens, Greece: National Technical University of Athens - Department of Electrical and Computer Engineering.

Panajotovic, Boban et al. (2011). "ICT and smart grid". In: *2011 10th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services, TELSIKS 2011 - Proceedings of Papers*, pp. 118–121. doi: 10.1109/TELSKS.2011.6112018.

Paul, Gerard (2005). *IPTraf - IP Network Monitoring Software*. [Online] Accessed 31 August 2018. url: http://iptraf.seul.org/.

Pires, P. et al. (2007). "Security aspects of SCADA and corporate network interconnection: An overview". In: *Proceedings of International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2006*, pp. 127–134. isbn: 0769525652. doi: 10.1109/DEPCOS-RELCOMEX.2006.46.

PLCScan (2012). *PLCScan*. [Online] Accessed 21 January 2018. url: http://code.google.com/p/plcscan.

Portworx (2017). *Portworx Annual Container Adoption Survey 2017*. [Online] Accessed 19 January 2018. url: https://portworx.com/wp-content/uploads/2017/04/Portworx_Annual_Container_Adoption_Survey_2017_Report.pdf.

Pyretic (2015). *Pyretic Language web site*. [Online] Accessed 19 January 2018. url: http://frenetic-lang.org/pyretic/.

Queiroz (2017). "Integration of SDN technologies in SCADA Industrial Control Networks". MA thesis. Coimbra, Portugal: University of Coimbra.

Raj Jain (2013). *Recent Advances in Networking - Data Center Virtualization, SDN, Big Data, Cloud Computing, Internet of Things*. http://www.cse.wustl.edu/~jain/cse570-13/. Accessed 22 December 2017.

Raymond, E. (2008). *The Cathedral & the Bazaar - Musings on Linux and Open Source by an Accidental Revolutionary*.

Regnell, Björn et al. (2008). "Supporting roadmapping of quality requirements". In: *IEEE Software* 25.2, pp. 42–47. issn: 07407459. doi: 10.1109/MS.2008.48.

Royce, Walker W. (1970). "Managing the development of large software systems: concepts and techniques". In: *Proc. IEEE WESTCON, Los Angeles*. Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 1987, pp. 328–338, pp. 1–9. url: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf.

Sadeghi, Ahmad-Reza et al. (2015). "Security and privacy challenges in industrial internet of things". In: *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*, pp. 1–6. isbn: 9781450335201. doi: 10.1145/2744769.2747942. url: http://dl.acm.org/citation.cfm?doid=2744769.2747942.

Sainz, Markel others (2017). "Software Defined Networking opportunities for intelligent security enhancement of Industrial Control Systems". In: *Proceedings in International Joint Conference SOCO'17-CISIS'17-ICEUTE'17 León, Spain, September 6–8, 2017,*

Salman, Ola et al. (2016). "SDN controllers: A comparative study". In: *Proceedings of the 18th Mediterranean Electrotechnical Conference: Intelligent and Efficient Technologies and Services for the Citizen, MELECON 2016*. isbn: 9781509000579. doi: 10.1109/MELCON.2016.7495430.

Scheitzer (2016). "SEL-2740S Software-Defined Network (SDN) Switch Traffic-Engineered Ethernet Communication for Substation and Plant Networks Major Features and Benefits". In: *Product Specification*.

SDNCentral (2016). *Special Report: OpenFlow and SDN – State of the Union*. Special report. url: https://www.opennetworking.org/images/stories/downloads/sdn-resources/special-reports/Special-Report-OpenFlow-and-SDN-State-of-the-Union-B.pdf. ONF, On.Lab, SDxCentral.

SDxcentral (2016). *What is NFV – Network Functions Virtualization – Definition*. https://www.sdxcentral.com/nfv/definitions/whats-network-functions-virtualization-nfv/. Accessed 15 January 2018.

Seetharaman, Srini (2015). *OpenDayLight: App development tutorial*. https://pt.slideshare.net/sdnhub/opendaylight-app-development-tutorial. Accessed 11 June 2017.

Shah, Syed Abdullah et al. (2013). "An architectural evaluation of SDN controllers". In: *IEEE International Conference on Communications*, pp. 3504–3508. isbn: 9781467331227. doi: 10.1109/ICC.2013.6655093.

Shalimov, Alexander et al. (2013). "Advanced Study of SDN / OpenFlow controllers". In: *Proceeding CEE-SECR '13 Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, pp. 1–6. doi: 10.1145/2556610.2556621.

Shanmugam, P K et al. (2014). "DEIDtect: Towards distributed elastic intrusion detection". In: *DCC 2014 - Proceedings of the ACM SIGCOMM 2014 Workshop on Distributed Cloud Computing*, pp. 17–23. doi: 10.1145/2627566.2627579.

Sharma, Abhinav et al. (2016). "SDN in SCADA Based System for Power Utilities: A Case Study of Himachal Pradesh State Electricity Board Limited SCADA System". In: *Indian Journal of Science and Technology* 9.32. issn: 0974 -5645. url: http://52.172.159.94/index.php/indjst/article/view/100220.

Sieber, Christian et al. (2016). "Towards a programmable management plane for SDN and legacy networks". In: *IEEE NETSOFT 2016 - 2016 IEEE NetSoft Conference and Workshops: Software-Defined Infrastructure for Networks, Clouds, IoT and Services*, pp. 319–327. isbn: 9781467394864. doi: 10.1109/NETSOFT.2016.7502428.

SonarSource (2018). *The leading product for Continuous Code Quality*. [Online] Accessed 31 August 2018. url: https://www.sonarqube.org/.

Subramanian, S. et al. (2016). *Software-Defined Networking (SDN) with OpenStack*. Packt Publishing. isbn: 9781786465993. url: https://books.google.pt/books?id=g%5C_UNvgAACAAJ.

Taima (2014). *SE- 565 Software System Requirements III. Requirements Elicitation*. https://www.slideserve.com/taima/se-565-software-system-requirements-iii-requirements-elicitation. Accessed 4 July 2017.

Taylor, P. and R. Mead (2016). *Delivering Successful PMOs: How to Design and Deliver the Best Project Management Office for Your Business*. Taylor & Francis. isbn: 9781317153283. url: https://books.google.pt/books?id=-7a1CwAAQBAJ.

Techtarget (2017). *The Basics of Ladder Logic*. http://whatis.techtarget.com/definition/intelligent-electronic-device. [Online] Accessed 15 January 2018.

TechTargetDef (2017). *Ternary content-addressable memory (TCAM)*. [Online] Accessed 25 December 2017. url: http://searchnetworking.techtarget.com/definition/TCAM-ternary-content-addressable-memor.

Thenewstack (2015). *OpenDaylight Licensing*. https://thenewstack.io/sdn-series-part-vi-opendaylight/. Accessed 15 June 2017.

Thompson, L.M. (2007). *Industrial Data Communications*. Resources for Measurement and Control Series. ISA. isbn: 9781934394243. url: https://books.google.pt/books?id=N8IvCes1o4cC.

Tinsley, S. et al. (2006). *Environmental Management Systems: Understanding Organizational Drivers and Barriers*. Earthscan. isbn: 9781853839368. url: https://books.google.pt/books?id=LrbtAAAAMAAJ.

TLF (2018). *DPDK official website*. [Online] Accessed 21 January 2018. url: https://dpdk.org/.

Tootoonchian, Amin et al. (2012). "On controller performance in software-defined networks". In: *Proceeding Hot-ICE'12 Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, pp. 10–10. issn: 0740-7475. doi: 10.1145/2491185.2491199. url: https://www.usenix.org/system/files/conference/hot-ice12/hotice12-final33%7B%5C_%7D0.pdf.

Trandafir, Ruxandra et al. (2016). "HoneYDSPK: Cisco onePK implementation for anomaly-based IDS and honeypot services". In: *Networking in Education and Research: RoEduNet International Conference 15th Edition, RoEduNet 2016 - Proceedings*. isbn: 9781509053988. doi: 10.1109/RoEduNet.2016.7753221.

Trema (2018). *Cbench source code repository*. https://github.com/trema/cbench. Accessed 4 January 2018.

Trois, Celio et al. (2016). *A Survey on SDN Programming Languages: Toward a Taxonomy*. doi: 10.1109/COMST.2016.2553778.

Vuex (2018). *Vuex library*. [Online] Accessed 21 January 2018. url: https://vuex.vuejs.org/.

Wang, Haopei et al. (2017). "Bring Your Own Controller : Enabling Tenant-defined SDN Apps in IaaS Clouds". In:

Wavestone (2018). *A low-cost, DIY data diode for ICS*. [Online] Accessed 21 January 2018. url: https://github.com/wavestone-cdt/dyode.

Whatpixel.com (2016). *Vue-JS learning resources*. http://whatpixel.com/images/2016/vuejs-learning-resources/. Accessed 4 January 2018.

Williams, James (2006). *Non-Functional Requirements List*. http://tynerblain.com/blog/2006/05/05/non-functional-requirements-list/. Accessed 3 July 2017.

Wired (2014). *An unprecedented look at Stuxnet, the world's first digital weapon*. [Online] Accessed 16 January 2018. url: https://www.wired.com/2014/11/countdown-to-zero-day-stuxnet/.

Y., Zhang et al. (2017). "A Communication-Aware Container Re-Distribution Approach for High Performance VNFs". In: *Proceedings - International Conference on Distributed Computing Systems*, pp. 1555–1564. isbn: 9781538617915. doi: 10.1109/ICDCS.2017.10.

Yoon, Changhoon et al. (2015). "Enabling security functions with SDN: A feasibility study". In: *Computer Networks* 85, pp. 19–35. issn: 13891286. doi: 10.1016/j.comnet.2015.05.005.

Zhang, Yang et al. (2017). "When Raft Meets SDN". In: *Proceedings of the First Asia-Pacific Workshop on Networking - APNet'17*, pp. 1–7. doi: 10.1145/3106989.3106999. url: http://dl.acm.org/citation.cfm?doid=3106989.3106999.

Zhao, Jinjing et al. (2016). "Network-Aware QoS Routing for Smart Grids Using Software Defined Networks". In: *LNICST* 166, pp. 384–394. issn: 0717-6163. doi: 10.1007/978-

3-319-33681-7_32. arXiv: 9809069v1 [arXiv:gr-qc]. url: http://dx.doi.org/10.1007/978-3-319-33681-7%7B%5C_%7D32.

Zhao, Yimeng et al. (2016). "On the performance of SDN controllers: A reality check". In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network, NFV-SDN 2015*, pp. 79–85. isbn: 9781467368841. doi: 10.1109/NFV-SDN.2015.7387410.

Zhao, Zheng et al. (2017). "An SDN-based fingerprint hopping method to prevent fingerprinting attacks". In: *Security and Communication Networks* 2017. issn: 19390122. doi: 10.1155/2017/1560594.

Master's Degree in Informatics Engineering
Volume II - Dissertation Annexes

Network Softwarization for IACS Security Applications

Miguel Rosado Borges de Freitas

miguelbf@student.dei.uc.pt

Supervisor: Prof. Doutor Tiago Cruz
Co-Supervisor: Prof. Doutor Paulo Simões

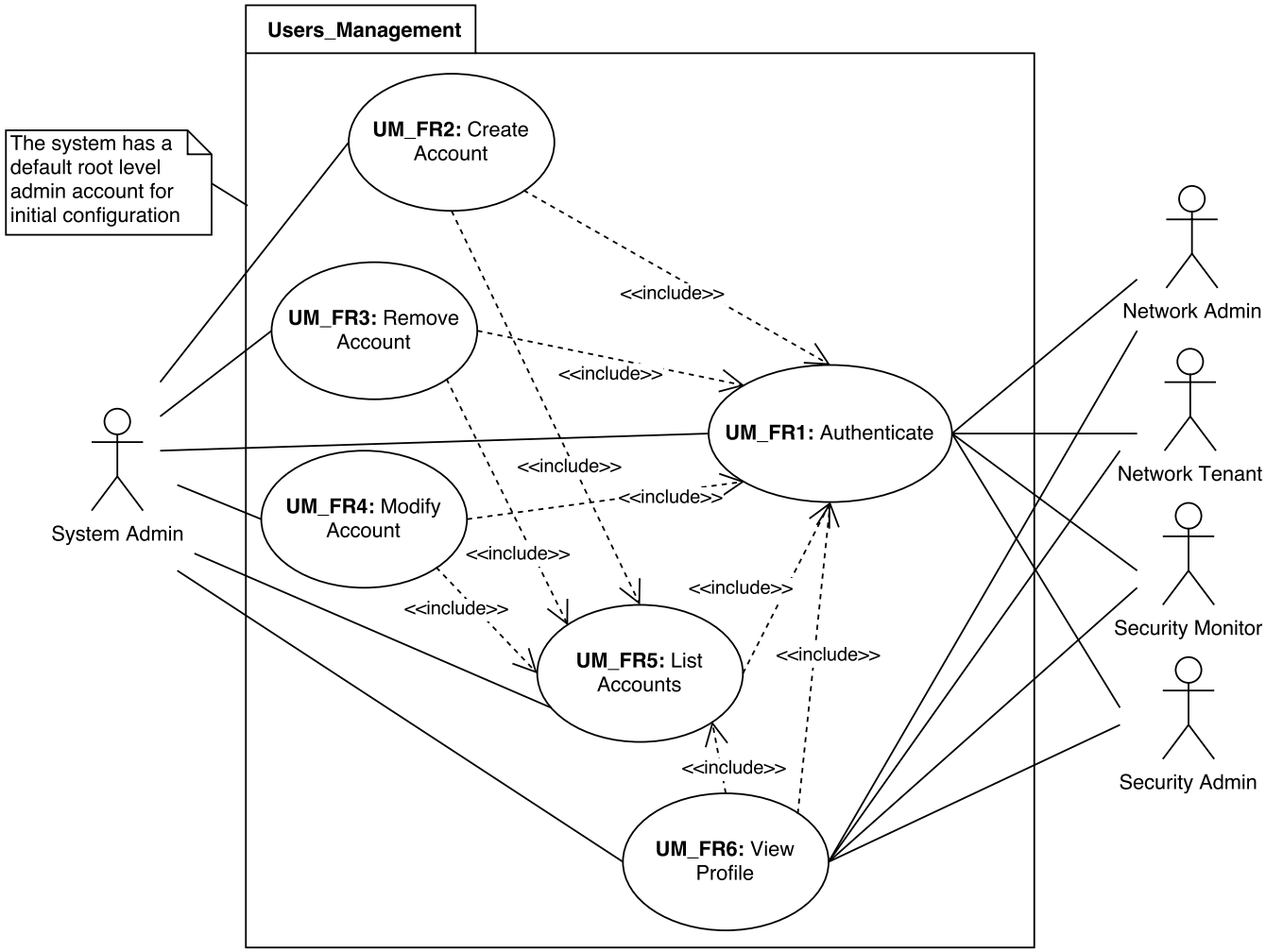
Coimbra, Friday 31st August, 2018

FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

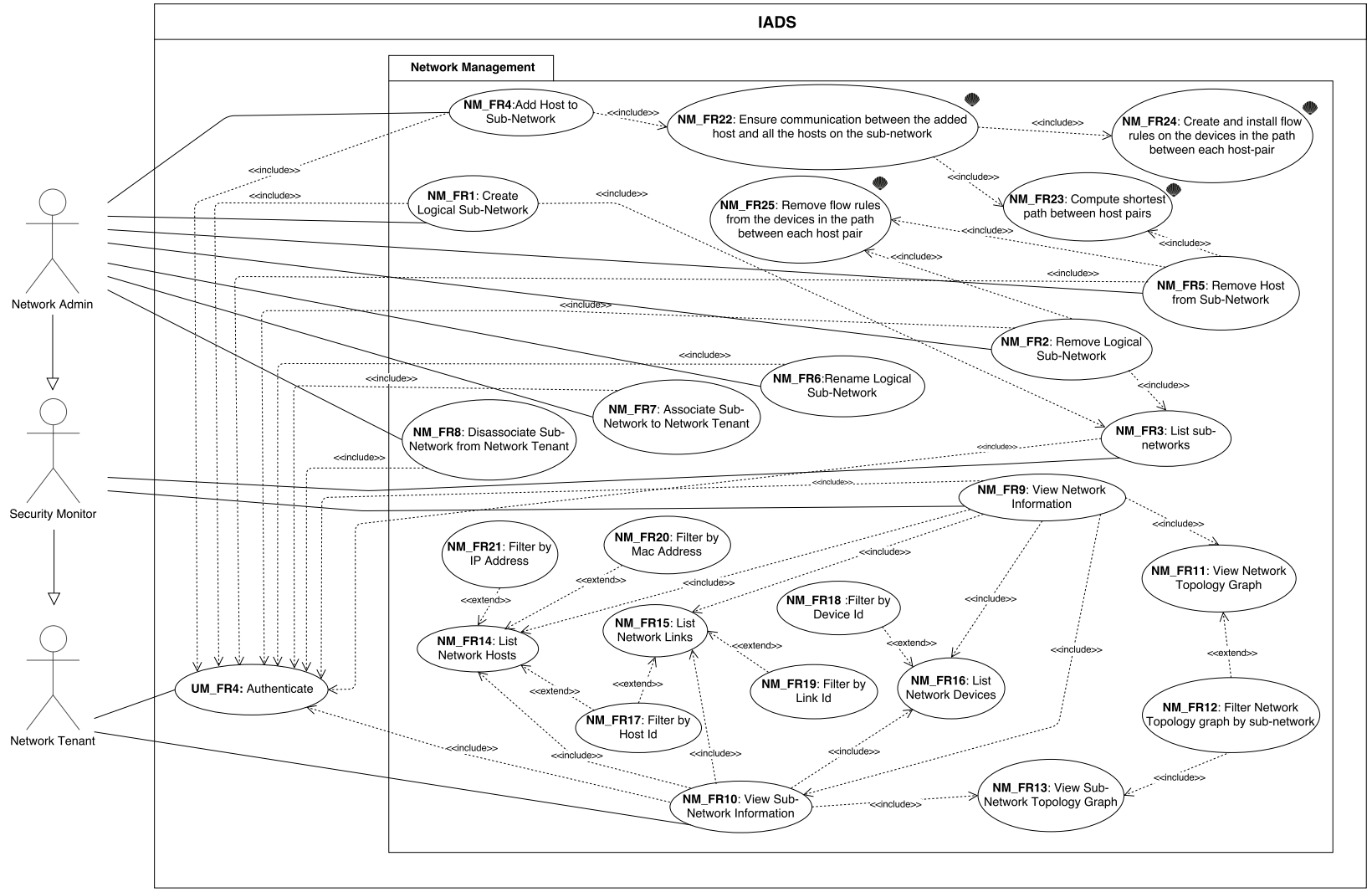
Contents

Appendix A

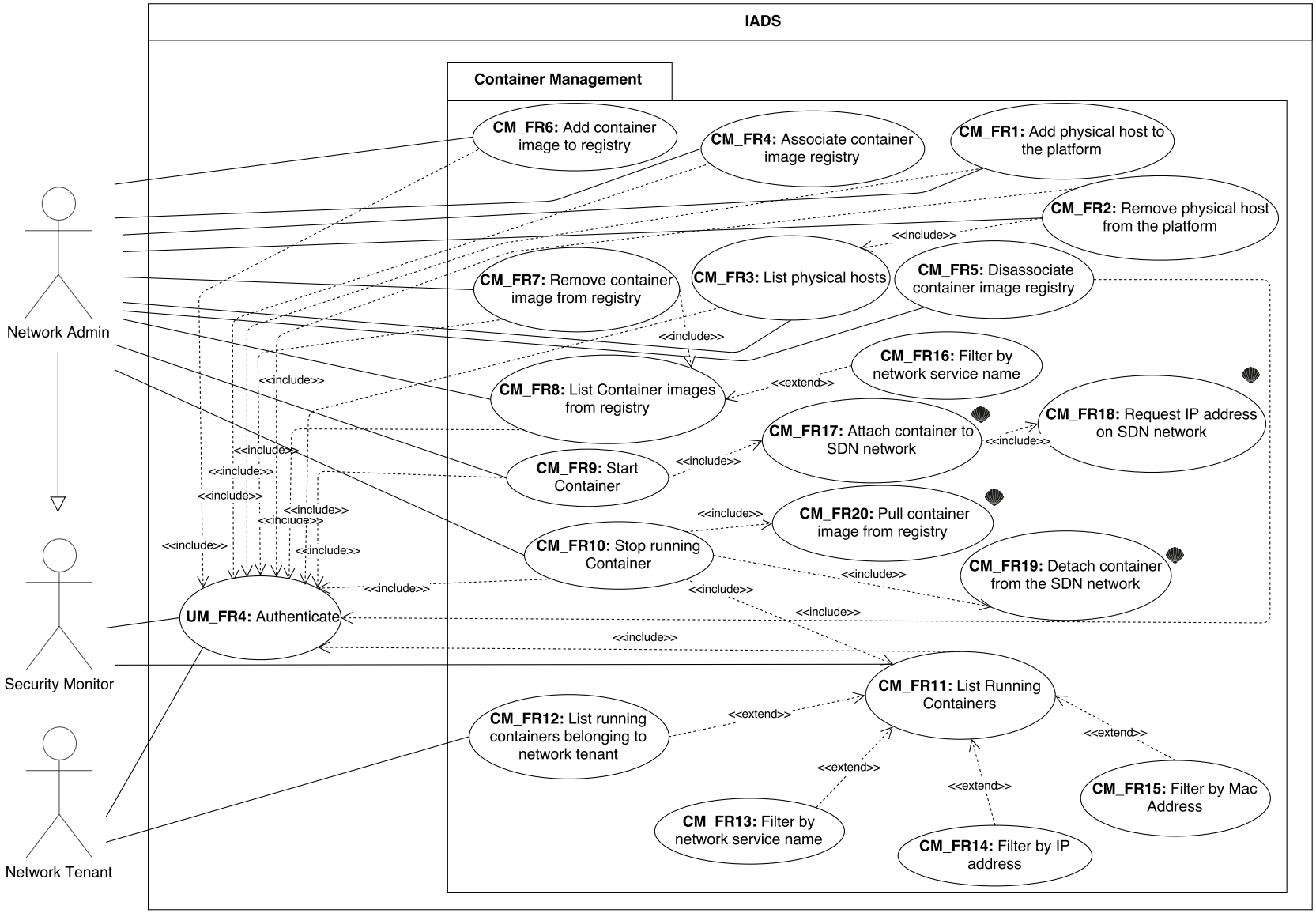
Use-case diagrams



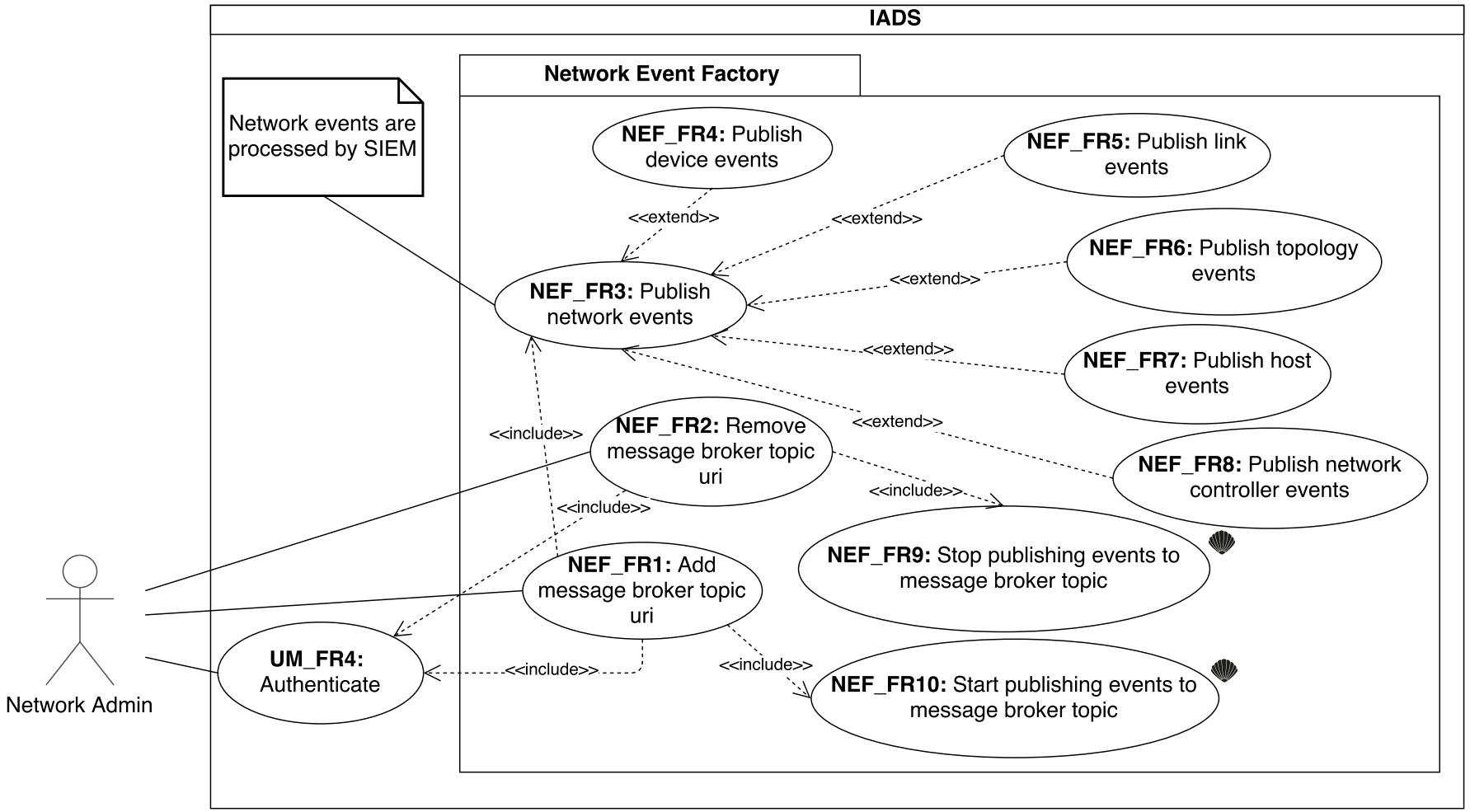
UC Diagram 1: Users_Management use case package



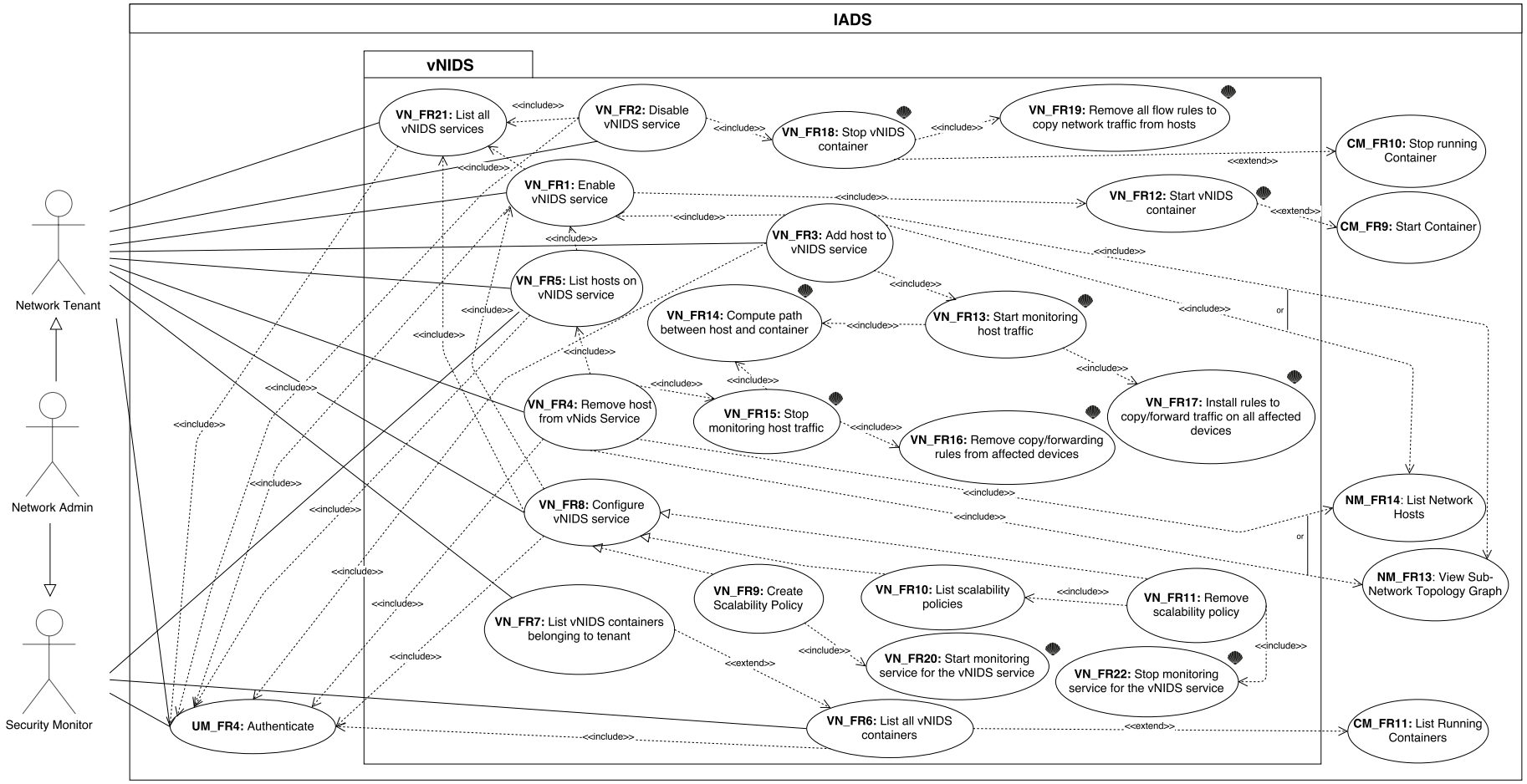
UC Diagram 2: Network_Management use case package



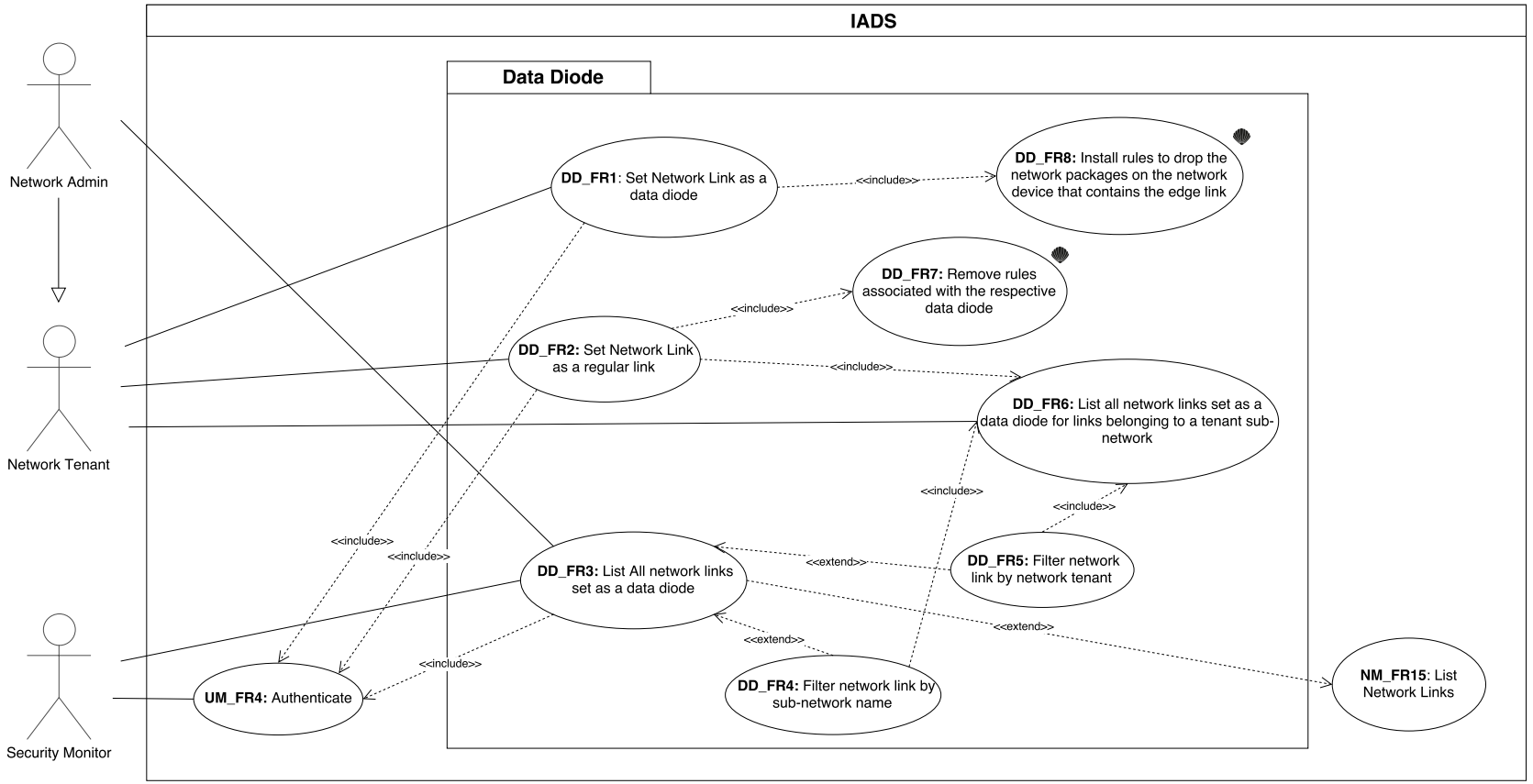
UC Diagram 3: Container_Management use case package



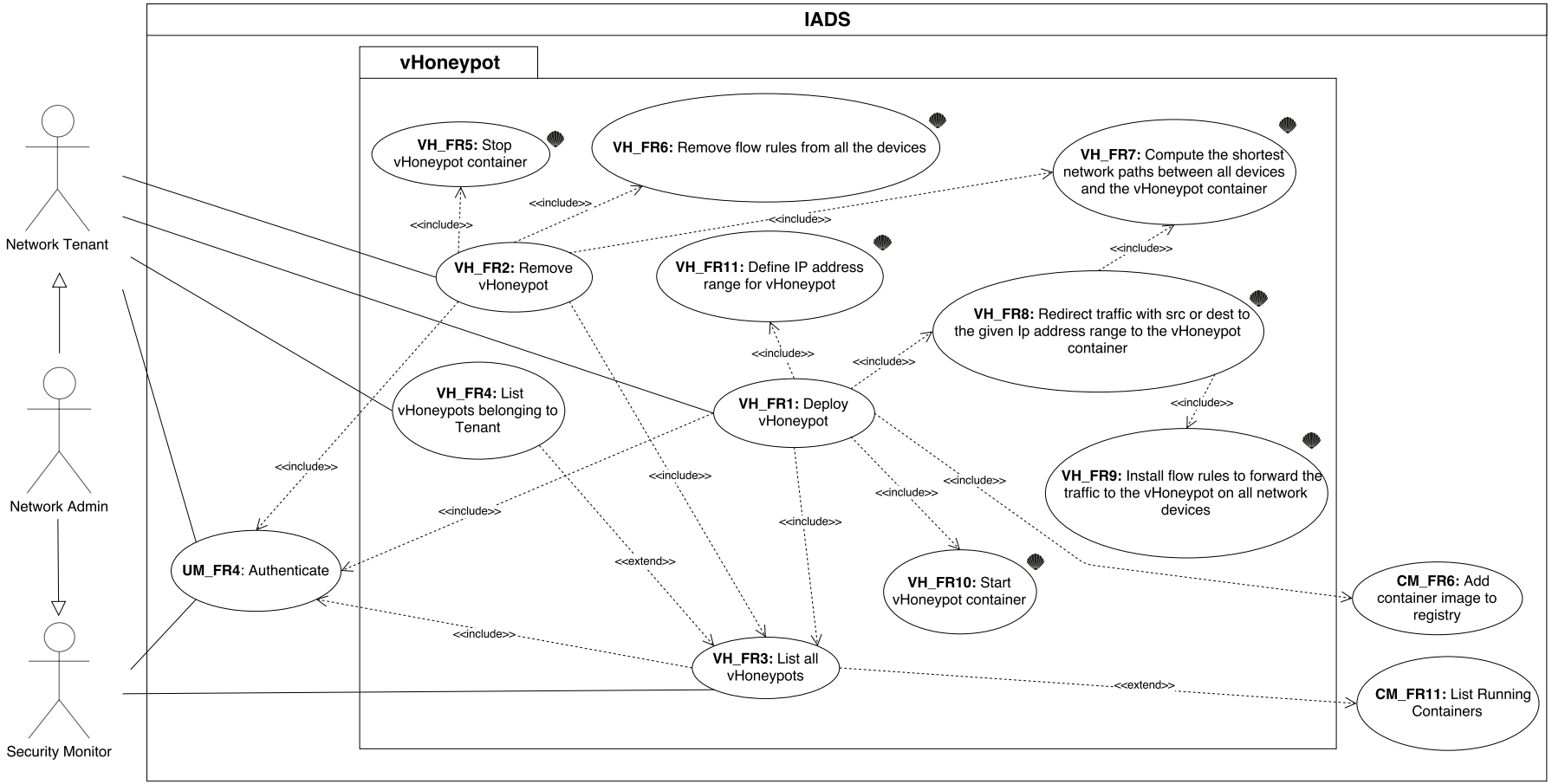
UC Diagram 4: Network_Event_Factory use case package



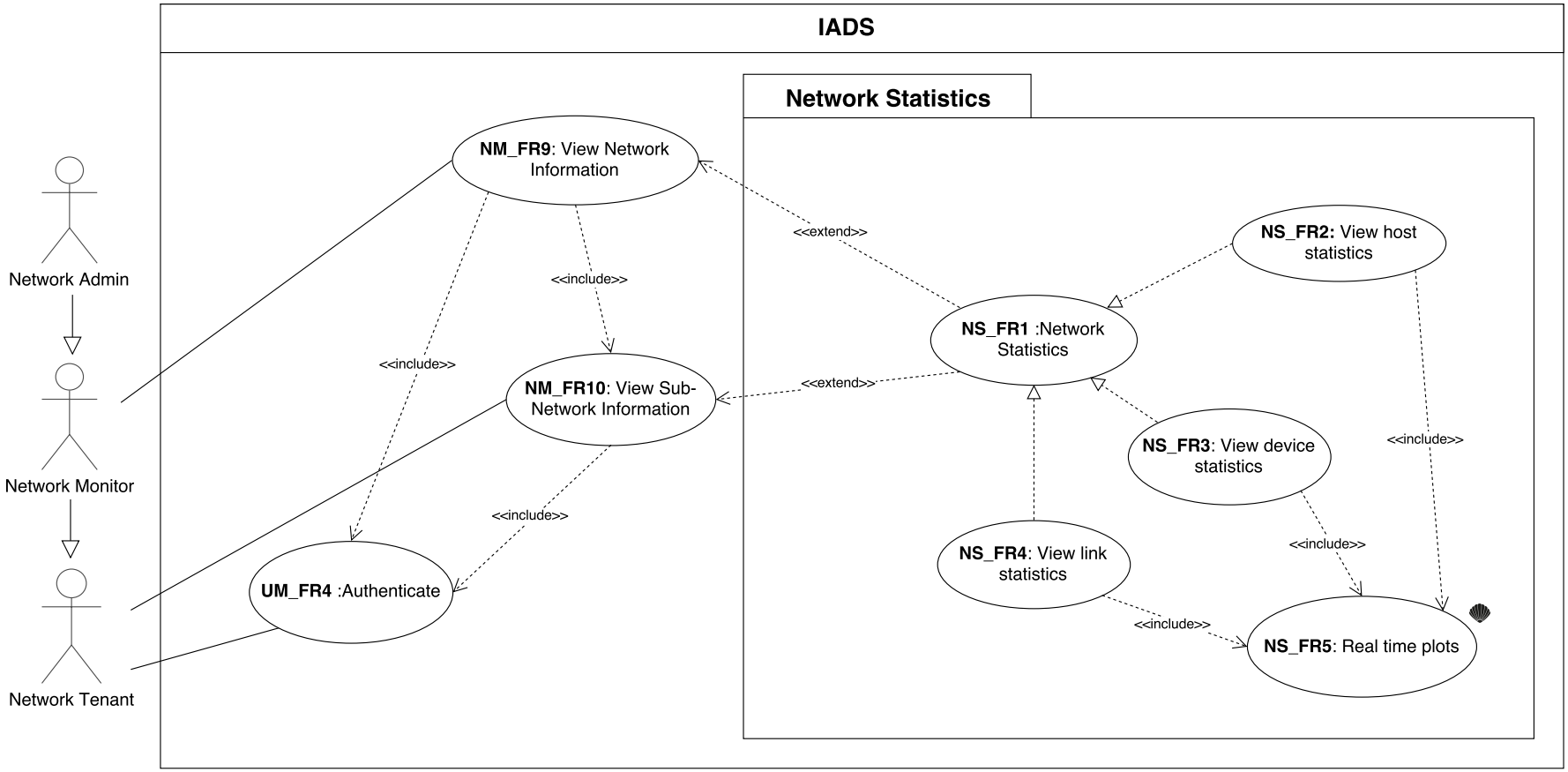
UC Diagram 5: vNIDS use case package



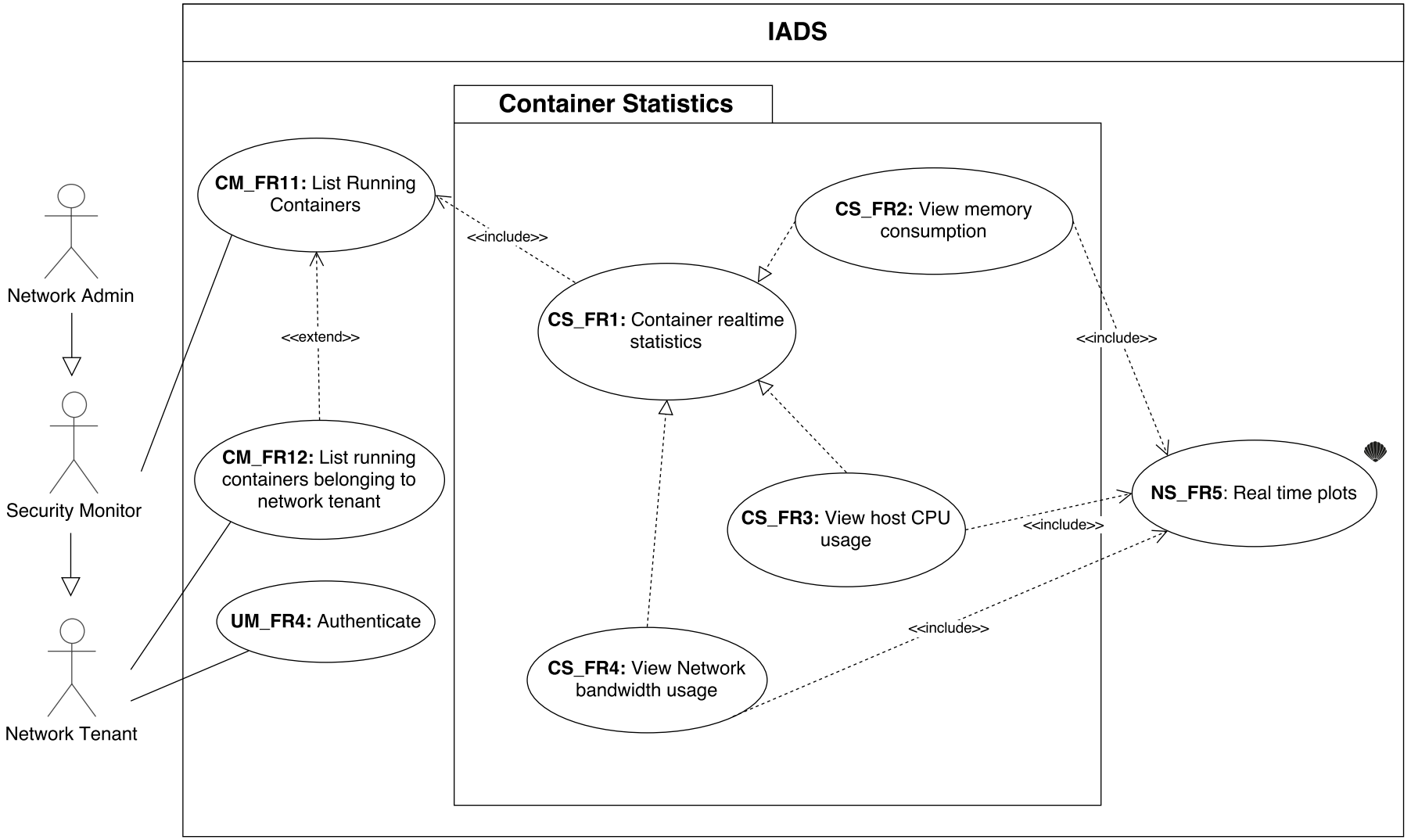
UC Diagram 6: Data diode use case package



UC Diagram 7: vHoneypot use case package



UC Diagram 8: Network_Statistics use case package



UC Diagram 9: Container_Statistics use case package

Appendix B

Use-case descriptions

B.1 Management Package

B.1.1 Users_Management package

Table B.1: Use case: UM_FR1 - Authenticate

Use Case	UM_FR1: Authenticate
Primary Actor	System Admin
Secondary Actors	Security Monitor, Security Admin, Network Admin, Network Tenant
Scope	IADS (system) - Black-box
Level	System goal
Stakeholders and Interests	– **System Admin**: interest in ensuring the segregation of functionalities – **Security Admin**: interest in being able to perform the features that his/her level of permissions grants – **Security Monitor**: interest in being able to perform the features that his/her level of permissions grants – **Network Admin**: interest in being able to perform the features that his/her level of permissions grants – **Network Tenant**: interest in being able to perform the features that his/her level of permissions grants
Pre-Conditions	– The website is available
Minimum Guarantees	An error message is presented
Success Guarantees	The actor is successfully authenticated
Trigger	Actor (unauthenticated) accesses the platform

Process - Main Success Scenario	1. Actor (unauthenticated) accesses the platform and it is redirected to login page view 2. Actor fills corresponding username and password 3. Actor confirms the action by clicking "Login" 4. Actor is authenticated by the system 5. Actor is redirected to the home page view
Exceptions	2. (a) The username and password are incorrect i. The system presents a popup message *"Your username or password are incorrect, please try again!"* ii. The use case ends (b) The account does not exist i. The system presents a popup message *"This account does not exist!"* ii. The use case ends 3. (a) Actor does not confirm the authentication action i. The use case ends 4. (a) A backend occurred i. The system presents a popup message *"An error occurred and the previous operation was not executed! Please try again later."* and the exception is logged ii. The use case ends

Table B.2: Use case: UM_FR2: Create Account

Use Case	**UM_FR2: Create Account**
Primary Actor	System Admin
Scope	IADS (system) - Black-box
Level	User goal
Stakeholders and Interests	– **System Admin**: interest in ensuring the segregation of functionalities – **Security Admin**: interest in having an account to perform the features that his/her level of permissions grants – **Security Monitor**: interest in having an account to perform the features that his/her level of permissions grants – **Network Admin**: interest in having an account to perform the features that his/her level of permissions grants – **Network Tenant**: interest in having an account to perform the features that his/her level of permissions grants
Pre-Conditions	– The website is available – Actor must be authenticated as System Admin via use case UM_FR1
Minimum Guarantees	An error message is presented

Success Guarantees	The account is successfully created
Trigger	Actor clicks on *"Create Account"*
Process - Main Success Scenario	1. Actor is redirected to the view *"Create account"* 2. Actor selects which kind of account wants to create and also fills all required fields 3. Actor clicks on *"Create Account"* and a confirmation modal is opened 4. Actor confirms the action by clicking *"Yes"* 5. A new account is created 6. The system displays a success message
Exceptions	4. (a) Actor does not confirm the action by clicking *"Yes"* i. The use case ends 5. (a) A backend error occurs i. The system presents a popup message *"An error occurred and the previous operation was not executed! Please try again!"* and the exception is logged ii. The use case ends

Table B.3: Use case: UM_FR3: Remove Account

Use Case	**UM_FR3: Remove Account**
Primary Actor	System Admin
Scope	IADS (system) - Black-box
Level	User goal
Stakeholders and Interests	– **System Admin**: interest in deleting a specific account
Pre-Conditions	– The website is available – Actor must be authenticated as System Admin via use case UM_FR1 – Actor has completed use case UM_FR5
Minimum Guarantees	An error message is presented
Success Guarantees	An existing account is removed from the platform
Trigger	Actor selects the account to remove

Process - Main Success Scenario	1. Actor selects the account to remove by clicking on *"Remove Account"* and a confirmation modal is opened 2. Actor confirms the account removal by clicking on *"Yes"* 3. The account is removed 4. A success message is displayed by the system
Exceptions	2. (a) Actor does not confirm the action by clicking *"No"* i. The use case ends 3. (a) A backend error occurs i. The system presents a popup message *"An error occurred and the previous operation was not executed! Please try again!"* and the exception is logged ii. The use case ends

Table B.4: Use case: UM_FR4: Modify Account

Use Case	**UM_FR4: Modify Account**
Primary Actor	System Admin
Secondary Actors	Security Monitor, Security Admin, Network Admin, Network Tenant
Scope	IADS (system) - Black-box
Level	User goal
Stakeholders and Interests	– **System Admin**: interest in modifying the information for a specific account – **Security Monitor**: interest in modifying own account information – **Security Admin**: interest in modifying own account information – **Network Admin**: interest in modifying own account information – **Network Tenant**: interest in modifying own account information
Pre-Conditions	– The website is available – Actor must be authenticated via use case UM_FR1 – Actor has completed use case UM_FR5 if the actor is the System Admin – Actor has completed use case UM_FR5 if the actor is the Network Admin and the account to be modified belongs to a Network Tenant
Minimum Guarantees	An error message is presented
Success Guarantees	The account is modified

Trigger	Actor clicks on *"Modify Account"*
Process - Main Success Scenario	1. Actor changes all the fields that he intends to modify 2. Actor confirms the action by clicking on *"Save"* 3. The account information is modified 4. The system displays a success message
Exceptions	2. (a) Actor does not confirms the action of save modifications i. The use case ends (b) The actor leaves a required field empty i. The system presents a popup message *"Field X is required. Please review the provided information"* (c) The actor has not changed any field i. The system presents a popup message *"There are no fields to be updated"*. 3. (a) A backend error occurs i. The system presents a popup message *"An error occurred and the previous operation was not executed! Please try again!"* and the exception is logged ii. The use case ends

Table B.5: Use case: UM_FR5: List Accounts

Use Case	**UM_FR5: List Accounts**
Primary Actor	System Admin
Secondary Actors	Network Admin
Scope	IADS (system) - Black-box
Level	User goal
Stakeholders and Interests	– **System Admin**: interest in listing all platform accounts – **Network Admin**: interest in listing all platform accounts belonging to Network Tenants
Pre-Conditions	– The website is available – Actor must be authenticated as System Admin or Network Admin via use case UM_FR1
Minimum Guarantees	An error message is presented
Success Guarantees	The system presents a list of accounts registered in the system
Trigger	Actor clicks on the *"Users"* item in the website menu

Process - Main Success Scenario	1. Actor clicks on the "Users" item in the website menu 2. Actor views a list of registered users. The list contains: (a) All the users registered in the platform if the actor is the System Admin (b) The list of Network Tenants if the actor is a Network Admin
Exceptions	2. (a) There are no registered users in the platform i. The system displays a message *"There are no registered users in the platform"* ii. The use case ends (b) A backend error occurs i. The system presents a popup message *"An error occurred and the previous operation was not executed! Please try again!"* and the exception is logged ii. The use case ends

Table B.6: Use case: UM_FR6: View Profile

Use Case	UM_FR6: View Profile
Primary Actor	System Admin
Secondary Actors	Security Monitor, Security Admin, Network Admin, Network Tenant
Scope	IADS (system) - Black-box
Level	User goal
Stakeholders and Interests	– **System Admin**: interest in viewing his profile data and the account data of all the users on the platform – **Network Admin**: interest in viewing his profile data and the account data of all the network tenants – **Network Tenant, Security Monitor, Security Admin**: interest in viewing his profile data
Pre-Conditions	– The website is available – Actor must be authenticated via use case UM_FR1 – Actor has completed use case UM_FR5 if the actor is a Network Admin or a System admin and intends to view a profile belonging to other user
Minimum Guarantees	An error message is presented
Success Guarantees	The user profile is displayed
Trigger	Actor clicks on *"View Profile"*

Process - Main Success Scenario	1. Actor clicks on "View profile". This can be accomplished: (a) From the navigation bar of the website for any user (b) From the list of registered users displayed by use case UM_FR5 if the actor is a Network Admin or a System Admin 2. The system displays all the information associated with the selected profile. This includes: (a) All the data provided when the user was registered on the system via use case UM_FR2 (b) All the account data the user might have modified via use case UM_FR4 (c) Any other information or assets associated with the user
Exceptions	2. (a) A backend error occurs i. The system presents a popup message *"An error occurred and the previous operation was not executed! Please try again!"* and the exception is logged ii. The use case ends

B.1.2 Network_Management package

Table B.7: Use case: NM_FR1: Create Logical Sub-Network

Use Case	**NM_FR1: Create Logical Sub-Network**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in creating logical sub-sections of the overall network – **Network Tenant**: interest in managing his own logical section of the overall network
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Admin via use case UM_FR1 – The actor is in the networks list via use case NM_FR3
Minimum Guarantees	An error message is presented
Success Guarantees	Logical sub-network is successfully created
Trigger	Actor clicks on *"Create new network"*

Process - Main Success Scenario	1. Actor selects *"Create new network"* and a modal is opened 2. Actor fills the name of the sub-network 3. Actor confirms the creation of the sub-network by clicking *"Save"* 4. The list of available sub-networks is updated and contains the created sub-network and a success message is shown
Exceptions	3. (a) The system already contains a network with the same name i. The system presents a popup message *"A network with the same name already exists"* and the modal is kept open (b) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (c) Actor dismisses the modal without clicking save i. User is redirected to the sub-network list via use-case NM_FR3 and the sub-network creation process is aborted 4. (a) SDN controller/API is not available i. The system presents a popup message "Could not connect to the controller, please try again later" and the exception is logged

Table B.8: Use case: NM_FR2: Remove Logical Sub-Network

Use Case	**NM_FR2: Remove Logical Sub-Network**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in removing an existing sub-section of the network
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Admin via use case UM_FR1 – The actor is in the networks list via use case NM_FR3
Minimum Guarantees	An error message is presented
Success Guarantees	Logical sub-network is successfully removed
Trigger	Actor finds the sub-network to remove in the list of sub-networks

Process - Main Success Scenario	1. Actor finds the sub-network to remove in the list of sub-networks 2. Actor clicks on *"remove sub-network"* and a confirmation modal is opened 3. Actor confirms the sub-network removal by clicking on *"Yes"* 4. The system removes all host-pair OpenFlow rules from the network devices between each host-pair 5. A success message is shown 6. The list of available sub-networks is updated and no longer contains the removed sub-network
Exceptions	2. (a) The system could not remove the sub-network i. The system presents a popup message *"Could not remove the sub-network"* and provides the exception cause (b) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged 3. (a) Actor dismisses the modal by clicking *"NO"* i. User is redirected to the sub-network list and the sub-network removal process is aborted 4. (a) OpenFlow enabled device is not available i. Sub-network removal process is aborted and an error message is show. The exception is logged. 6. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.9: Use case: NM_FR3: List sub-networks

Use Case	**NM_FR3: List sub-networks**
Primary Actor	Network Admin
Secondary Actor	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in viewing the logical sub-sections of the overall network for management purposes – **Security Monitor**: interest in viewing the logical sub-sections of the overall network for monitoring purposes

Pre-Conditions	– The website is available – Actor must be authenticated as a Network Admin or as a Security Monitor via use case UM_FR1
Minimum Guarantees	An error message is presented
Success Guarantees	Logical sub-networks are listed
Trigger	Actor selects the network tab on the platform menu
Process - Main Success Scenario	1. Actor selects the Network tab on the platform menu 2. Actor is redirected to the Network List view
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.10: Use case: NM_FR4: Add Host to Sub-Network

Use Case	**NM_FR4: Add Host to Sub-Network**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in creating logical sub-sections of the overall network – **Security Tenant**: interest in having hosts on his own logical section of the overall network
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Admin via use case UM_FR1 – The actor is in a view containing a list of hosts. These can be: 1. Via use-case NM_FR11 (view network topology graph) 2. Via use-case NM_FR14 (list network hosts)
Minimum Guarantees	An error message is presented
Success Guarantees	Host is added to the sub-network
Trigger	Actor clicks on *"Add Host to Sub-Network"*

Process - Main Success Scenario

1. Actor clicks on "Add Host to Sub-Network". This can be achieved by:
   (a) Clicking on a network host on the topology graph (via UC NM_FR11) and the corresponding button on the tools panel related to selected host
   (b) Clicking on the button that is presented on the list provided by UC NM_FR14
2. The system presents a confirmation dialog along with the message *"Are you sure you want to add host x to sub-network y?"*
3. Actor confirms the action by clicking on *"Yes"*
4. The system ensures the communication between the host to be added and all the hosts that were already on the sub-network (NM_FR22) by:
   (a) Computing the shortest path (network devices and respective ports) between the added host and all the hosts already on the sub-network (sub-function NM_FR23)
   (b) Creating host-pair rules for each device using the Mac Addresses of the host pair and the ingress and egress ports of the network device (sub-function NM_FR24)
   (c) Install the created rules on all devices on the path (sub-function NM_FR24)
5. The system presents a success message
6. The information provided by use case NM_FR10 (View Sub-Network Information) is updated. This includes:
   (a) The added host shows in the sub-network topology graph (use case NM_FR13)
   (b) The added host shows in the list of sub-network hosts (use case NM_FR14)

Exceptions

1.
3.
6. (a) SDN controller/API is not available
    i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged
3. (a) Actor dismisses the modal without clicking save
    i. User is redirected to the sub-network list via use-case NM_FR3 and the sub-network creation process is aborted
4. (a) An OpenFlow enabled device on a path between an host-pair is not available before the host is added to the sub-network
    i. The system computes an alternate path and the use case continues
   (b) An OpenFlow enabled device on a path between an host-pair is not available before the host is added to the sub-network and there are no alternative paths between the host-pair
    i. An error message is shown *"There is no path between host x and y"*, the exception is logged and the UC terminates
   (c) An OpenFlow enabled device on a path between an host-pair goes down after the host is added to the network
    i. The system tries to find an alternative path between the hosts and recomputes and install alternative OpenFlow rules
   (d) An OpenFlow enabled device on a path between an host-pair goes down after the host is added to the network and there is no alternative paths between hosts
    i. An alert is displayed to the user *"There is an outage on the communication between x and y because network device z is offline"* the exception is logged and the use case ends

Table B.11: Use case: NM_FR5: Remove Host from Sub-Network

Use Case	**NM_FR5: Remove Host from Sub-Network**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in changing existing logical sub-sections of the overall network
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Admin via use case UM_FR1 – The actor is in a view containing a list of hosts. These can be: 1. Via use-case NM_FR11 (view network topology graph) 2. Via use-case NM_FR13 (view sub-network topology graph) 3. Via use-case NM_FR14 (list network hosts)
Minimum Guarantees	An error message is presented
Success Guarantees	Host is removed from the sub-network
Trigger	Actor clicks on *"Remove Host from Sub-Network"*

Process - Main Success Scenario

1. Actor clicks on **"Remove Host from Sub-Network"**. This can be achieved by:
   (a) Clicking on a network host on the topology graph (via UC NM_FR11 or via UC NM_FR13) and the corresponding button on the tools panel related to selected host
   (b) Clicking on the button that is presented on the list provided by UC NM_FR14
2. The system presents a confirmation dialog along with the message *"Are you sure you want to remove host x from sub-network y?"*
3. Actor confirms the action by clicking on *"Yes"*
4. The system ensures there is no communication between the removed host and all the other hosts on the sub-network. Thus, the system:
   (a) Computes the shortest path between the host all all the other hosts on the sub-network (sub-function NM_FR23)
   (b) For all devices in the path, remove the flow rules containing Mac Address of the host to be removed from the network (sub-function NM_FR25)
5. A success message is presented to the user
6. The information provided by use case NM_FR10 (View Sub-Network Information) is updated. This includes:
   (a) The removed host no longer shows in the sub-network topology graph (use case NM_FR13)
   (b) The removed host no longer shows in the list of sub-network hosts (use case NM_FR14)

Exceptions

1.
3.
6. (a) SDN controller/API is not available
   i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged
3. (a) Actor dismisses the modal without clicking *"Yes"*
   i. The sub-network creation process is aborted
4. (a) An OpenFlow enabled device on a path between an host-pair is not available before the host is removed from the sub-network
   i. The system stores the action so it is applied when the device is back

Table B.12: Use case: NM_FR7: Associate Sub-Network to Network Tenant

Use Case	**NM_FR7: Associate Sub-Network to Network Tenant**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in delegating management of a logical section of the overall network – **Network Tenant**: interest in managing a logical section of the overall network
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Admin via use case UM_FR1 – The sub-network exists and was created via use case NM_FR1 – The network tenant account exists and was created via use case UM_FR2 – The actor is in the networks list via use case NM_FR3, in the users list via use case UM_FR5 or in a specific user profile via use case UM_FR6
Minimum Guarantees	An error message is presented
Success Guarantees	Logical sub-network is associated with a network tenant
Trigger	Actor clicks on *"Associate Sub-Network to Network Tenant"*
Process - Main Success Scenario	1. Actor selects *"Associate Sub-Network to Network Tenant"* and a modal is opened. The modal shows: (a) The list of available sub-networks that do not belong already to the given tenant if use case is fulfilled via use cases UM_FR5 or UM_FR6 (b) The list of network tenants that do not have an associated network if the use case is fulfilled via use case NM_FR3 2. A confirmation message is displayed by the system 3. The actor confirms the association between the network tenant and the sub-network by clicking *"Ok"* 4. A success message is displayed

Exceptions	1. 4. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged 3. (a) Actor dismisses the modal without clicking *"ok"* i. The use case terminates and the user is redirected to the view of the origin use case (see 1) 4. (a) An exception occurs in the SDN controller during the association i. The system presents a popup message containing the exception ii. The exception is logged

Table B.13: Use case: NM_FR8: Disassociate Sub-Network from Network Tenant

Use Case	NM_FR8: Disassociate Sub-Network from Network Tenant
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in removing the management privileges of a logical section of the overall network from a network tenant
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Admin via use case UM_FR1 – The sub-network exists and was created via use case NM_FR1 – The network tenant account exists and was created via use case UM_FR2 – The sub-network was associated to a given tenant via use case UM_FR7 – The actor is in the networks list via use case NM_FR3, in the users list via use case UM_FR5 or in a specific user profile via use case UM_FR6
Minimum Guarantees	An error message is presented
Success Guarantees	Logical sub-network is disassociated from the network tenant
Trigger	Actor clicks on *"Disassociate Sub-Network from Network Tenant"*

Process - Main Success Scenario	1. Actor selects *"Disassociate Sub-Network from Network Tenant"* and a modal is opened. The user may select this button from: (a) The network item in the sub-networks section of the tenant profile (if use case is fulfilled from use case UM_FR6) (b) The network item in list of sub-networks (if use case is fulfilled via use case UM_FR3) 2. A confirmation message is displayed by the system 3. The actor confirms the disassociation action by clicking *"Ok"* 4. A success message is displayed
Exceptions	1. 4. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged 3. (a) Actor dismisses the modal without clicking *"ok"* i. The use case terminates and the user is redirected to the view of the origin use case (see use case pre-conditions) 4. (a) An exception occurs in the SDN controller i. The system presents a popup message containing the exception ii. The exception is logged

Table B.14: Use case: NM_FR9: View Network Information

Use Case	**NM_FR9: View Network Information**
Primary Actor	Network Admin
Secondary Actor	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all the information regarding the network – **Security Monitor**: be able to monitor the network
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Admin or Security Monitor via use case UM_FR1
Minimum Guarantees	An error message is presented

Success Guarantees	Actor can access the network information
Trigger	Actor clicks on the Network item in the platform menu
Process - Main Success Scenario	1. Actor clicks on the network item in the platform menu 2. A sub-menu opens from where the actor can select on the following options (a) View network topology graph via use case NM_FR11 (b) List network devices via use case NM_FR16 (c) List network hosts via use case NM_FR14 (d) List network links via use case NM_FR15 (e) View sub-network information via use case NM_FR10 (f) View network statistics via use case NS_FR1
Exceptions	-

Table B.15: Use case: NM_FR10: View Sub-Network Information

Use Case	**NM_FR10: View Sub-Network Information**
Primary Actor	Network Admin
Secondary Actor	Security Monitor, Network Tenant
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all the information regarding all the logical sub-networks – **Security Monitor**: be able to monitor all the sub-networks – **Network Tenant**: be able to monitor his own sub-network
Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 – Actor must be authenticated as a Network Admin or Security Monitor via use case UM_FR1 – The actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can access the sub-network information
Trigger	Actor clicks on the Network item in the platform menu (if a network tenant) or a specific sub-network (if a network admin or security monitor)

Process - Main Success Scenario	1. Actor clicks on the Network item in the platform menu (if a network tenant) or a specific sub-network (if a network admin or security monitor) 2. A submenu opens from where the actor can select on the following options (a) View the sub-network topology graph via use case NM_FR13 (b) List the sub-network devices via use case NM_FR16 (c) List the sub-network hosts via use case NM_FR14 (d) List the sub-network links via use case NM_FR15 (e) View the sub-network statistics via use case NS_FR1
Exceptions	-

Table B.16: Use case: NM_FR11: View Network Topology Graph

Use Case	**NM_FR11: View Network Topology Graph**
Primary Actor	Network Admin
Secondary Actor	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have a centralized view of the network for administration purposes – **Security Monitor**: have a centralized view of the network for monitoring purposes
Pre-Conditions	– The website is available – The actor is authenticated as network admin or as security monitor via use case UM_FR1 – The actor has fulfilled use case NM_FR9
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the network topology graph
Trigger	Actor clicks on the "Network topology" item in the platform menu
Process - Main Success Scenario	1. Actor clicks on the "network topology" item in the platform menu 2. The actor is presented with the topology graph. (a) The network assets of the network can be distinguished by their icon (host, device, container)

Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.17: Use case: NM_FR12: Filter Network Topology graph by sub-network

Use Case	**NM_FR12: Filter Network Topology graph by sub-network**
Primary Actor	Network Admin
Secondary Actor	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have a global view of a given sub-network – **Security Monitor**: have a global view of a given sub-network
Pre-Conditions	– The website is available – The actor is authenticated as network admin or as security monitor via use case UM_FR1 – The actor has fulfilled use case NM_FR11
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the sub-network topology graph
Trigger	Actor checks the sub-network from the network list panel in the global network topology view (use case NM_FR11)
Process - Main Success Scenario	1. Actor checks the sub-network from the network list panel in the global network topology view (use case NM_FR11) 2. The actor is presented with the sub-network topology graph (use case NM_FR13)
Exceptions	1. 2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.18: Use case: NM_FR13: View Sub-Network Topology Graph

Use Case	**NM_FR13: View Sub-Network Topology Graph**
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have a global view of a given sub-network – **Security Monitor**: have a global view of a given sub-network – **Network Tenant**: have a global view of the the logical network the actor controls
Pre-Conditions	– The website is available – The actor is authenticated as a network admin, a network tenant or as security monitor via use case UM_FR1 – The sub-network does exist and was created via use-case NM_FR1 – The actor has fulfilled use case NM_FR11 in case he/she is a network admin or security monitor – The actor is a network tenant, the sub-network was associated with the actor profile via use case NM_FR7 and the actor has fulfilled use case NM_FR10
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the sub-network topology graph
Trigger	1. **Network Tenant**: actor clicks on the network topology sub-menu 2. **Network Admin and Security Monitor**: actor filters the topology graph via use case NM_FR12
Process - Main Success Scenario	1. Actor clicks on the network topology sub-menu (if a network tenant) or filters the global network topology graph via use case NM_FR12 (if the actor is a network admin or a security monitor) 2. The actor is presented with the sub-network topology graph
Exceptions	1. 2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.19: Use case: NM_FR14: List Network Hosts

Use Case	NM_FR14: List Network Hosts
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all network assets – **Security Monitor**: have access to all network assets for monitoring purposes – **Network Tenant**: have access to all assets on the actor sub-network
Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 (if applied to a sub-network) – The actor is authenticated as network admin or as security monitor via use case UM_FR1 – The actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the list of hosts belonging to the network (or sub-network)
Trigger	Actor clicks on the hosts item in the platform menu
Process - Main Success Scenario	1. Actor clicks on the hosts item in the platform menu 2. The actor is presented with a list of hosts
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.20: Use case: NM_FR15: List Network Links

Use Case	NM_FR15: List Network Hosts
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor

Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all network assets and their connections – **Security Monitor**: have access to all network assets and their connections for monitoring purposes – **Network Tenant**: have access to all assets and their connections on his/her sub-network
Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 (if applied to a sub-network) – The actor is authenticated as network admin or as security monitor via use case UM_FR1 or the actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the list of links belonging to a network (or sub-network)
Trigger	Actor clicks on the hosts item in the platform menu
Process - Main Success Scenario	1. Actor clicks on the links item in the platform menu 2. The actor is presented with a list of links
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.21: Use case: NM_FR16: List Network Devices

Use Case	**NM_FR16: List Network Devices**
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal

Stakeholders and Interests	– **Network Admin**: have access to all network assets – **Security Monitor**: have access to all network assets for monitoring purposes – **Network Tenant**: have access to all assets on the actor sub-network
Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 (if applied to a sub-network) – The actor is authenticated as network admin or as security monitor via use case UM_FR1 or the actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the list of devices belonging to a network (or sub-network)
Trigger	Actor clicks on the devices item in the platform menu
Process - Main Success Scenario	1. Actor clicks on the devices item in the platform menu 2. The actor is presented with a list of devices
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.22: Use case: NM_FR17: Filter by Host Id

Use Case	NM_FR16: List Network Devices
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all network assets – **Security Monitor**: have access to all network assets for monitoring purposes – **Network Tenant**: have access to all assets on the actor sub-network

Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 (if applied to a sub-network) – The actor is authenticated as network admin or as security monitor via use case UM_FR1 or the actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7 – The actor has fulfilled use case NM_FR14
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the list of hosts or links belonging to a network (or sub-network)
Trigger	Actor searches for a Host Id
Process - Main Success Scenario	1. Actor searches for a Host Id 2. The actor is presented with a list of filtered hosts (if use case starts via use case NM_FR14) or with a list of network links (if use case starts via use case NM_FR15)
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.23: Use case: NM_FR18: Filter by Device Id

Use Case	**NM_FR18: Filter by Device Id**
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all network assets – **Security Monitor**: have access to all network assets for monitoring purposes – **Network Tenant**: have access to all assets on his/her sub-network

Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 (if applied to a sub-network) – The actor is authenticated as network admin or as security monitor via use case UM_FR1 or the actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7 – The actor has fulfilled use case NM_FR16
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see a filtered list of network devices
Trigger	Actor searches for a device by device id
Process - Main Success Scenario	1. Actor fills the search input in the view provided by use case NM_FR16 with the device id 2. The actor is presented with a list of filtered network devices
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.24: Use case: NM_FR19: Filter by Link Id

Use Case	NM_FR19: Filter by Link Id
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all network assets and their connections – **Security Monitor**: have access to all network assets and their connections for monitoring purposes – **Network Tenant**: have access to all assets and their connections on his/her sub-network

Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 (if applied to a sub-network) – The actor is authenticated as network admin or as security monitor via use case UM_FR1 or the actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7 – The actor has fulfilled use case NM_FR15
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see a filtered list of network links
Trigger	Actor searches for a network link by link id
Process - Main Success Scenario	1. Actor fills the search input in the view provided by use case NM_FR15 with the link id 2. The actor is presented with a list of filtered network links
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.25: Use case: NM_FR20: Filter by Mac Address

Use Case	**NM_FR20: Filter by Mac Address**
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all network assets – **Security Monitor**: have access to all network assets for monitoring purposes – **Network Tenant**: have access to all assets on his/her sub-network
Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 (if applied to a sub-network) – The actor is authenticated as network admin or as security monitor via use case UM_FR1 or the actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7 – The actor has fulfilled use case NM_FR14

Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the list of hosts or links belonging to a network (or sub-network)
Trigger	Actor fills the search field of the view presented by NM_FR14 with the host Mac Address
Process - Main Success Scenario	1. Actor fills the search field of the view presented by NM_FR14 with the host Ip Address 2. The actor is presented with a list of filtered hosts
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.26: Use case: NM_FR21: Filter by IP Address

Use Case	**NM_FR21: Filter by IP Address**
Primary Actor	Network Admin
Secondary Actor	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: have access to all network assets – **Security Monitor**: have access to all network assets for monitoring purposes – **Network Tenant**: have access to all assets on his/her sub-network
Pre-Conditions	– The website is available – The sub-network exists and was created via use case NM_FR1 (if applied to a sub-network) – The actor is authenticated as network admin or as security monitor via use case UM_FR1 or the actor is authenticated as network tenant via use case UM_FR1 and the sub-network is associated to his profile via use case NM_FR7 – The actor has fulfilled use case NM_FR14
Minimum Guarantees	An error message is presented
Success Guarantees	Actor can see the list of hosts or links belonging to a network (or sub-network)

Trigger	Actor fills the search field of the view presented by NM_FR14 with the host Ip Address
Process - Main Success Scenario	1. Actor fills the search field of the view presented by NM_FR14 with the host Ip Address 2. The actor is presented with a list of filtered hosts
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

B.1.3 Container_Management

Table B.27: Use case: CM_FR1: Add physical host to the platform

Use Case	**CM_FR1: Add physical host to the platform**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: configure the virtual infrastructure – **Security Monitor**: have access to a virtual infrastructure for monitoring purposes – **Network Tenant**: have a virtual infrastructure to which he/she can deploy network services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – The user is in the view provided by use case CM_FR3
Minimum Guarantees	An error message is presented
Success Guarantees	Container physical host is associated with the platform
Trigger	Actor clicks on *"Add Physical Host"* button
Process - Main Success Scenario	1. The actor clicks on *"Add Physical Host"* 2. The actor fills the information regarding the physical host. This may include: (a) Host IP address (b) Host port (c) Private key 3. The actor clicks *"Save"* 4. The container physical host is associated with the platform 5. The system displays a success message

Exceptions	2. (a) Required information is missing i. The system presents a popup stating *"The field X is mandatory"* the exception is logged (b) Incorrect information was submitted by the actor i. The system presents a popup stating *"The field X contents are incorrect"* 2. (a) The actor does proceed / click on *"save"* i. The use case ends 4. (a) The system cannot connect to the container physical host i. The system presents an error message "Could not connect to container host" and the exception is logged (b) SDN controller/API is not available i. The system displays an error message stating *"Could not contact the controller/API"* (c) An exception is thrown while adding the host i. The system displays an error message containing the exception and the exception is logged

Table B.28: Use case: CM_FR2: Remove physical host from the platform

Use Case	**CM_FR2: Remove physical host from the platform**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: configure the virtual infrastructure – **Security Monitor**: have access to a virtual infrastructure for monitoring purposes – **Network Tenant**: have a virtual infrastructure to which he/she can deploy network services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – The physical host was previously added to the system via use case CM_FR1 – The user is in the view provided by use case CM_FR3
Minimum Guarantees	An error message is presented
Success Guarantees	Container physical host is removed from the platform
Trigger	Actor finds the physical host he/she wants to remove in the physical hosts list

Process - Main Success Scenario	1. Actor finds the physical host he/she wants to remove in the physical hosts list 2. Actor clicks on *"Remove Physical Host"* and a confirmation modal is open 3. Actor confirms his/her intention by clicking *"Yes"* 4. All the containers running on host are destroyed 5. The container physical host is removed from the platform 6. The system displays a success message
Exceptions	3. (a) The actor clicks on *"No"* i. The use case ends (b) Incorrect information was submitted by the actor i. The system presents a popup stating *"The field X contents are incorrect"* 5. (a) SDN controller/API is not available i. The system displays an error message stating *"Could not contact the controller/API"* (b) An exception is thrown while removing the host i. The system displays an error message containing the exception and the exception is logged

Table B.29: Use case: CM_FR3: List physical hosts

Use Case	**CM_FR3: List physical hosts**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: configure the virtual infrastructure – **Security Monitor**: have access to a virtual infrastructure for monitoring purposes – **Network Tenant**: have a virtual infrastructure to which he/she can deploy network services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1
Minimum Guarantees	An error message is presented
Success Guarantees	The system displays the list of physical hosts
Trigger	Actor clicks on the sub-menu item *"Nodes"* of the menu item *"Virtual Infrastructure"*

Process - Main Success Scenario	1. Actor clicks on the sub-menu item *"Nodes"* of the menu item *"Virtual Infrastructure"* 2. Actor is presented with the list of physical hosts that support the virtual infrastructure (i.e. the physical hosts where containers are deployed)
Exceptions	2. (a) SDN controller/API is not available i. The system displays an error message stating *"Could not contact the controller/API"* (b) An exception is thrown while getting the list i. The system displays an error message containing the exception and the exception is logged

Table B.30: Use case: CM_FR4: Associate container image registry

Use Case	CM_FR4: Associate container image registry
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: add a centralized repository for virtual images – **Network Tenant**: have a centralized repository from which the services images can be stored – **Security Monitor**: monitor the virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – A container image registry exists and was created externally to the system
Minimum Guarantees	An error message is presented
Success Guarantees	Container registry is successfully associated with the system
Trigger	Actor clicks on *"template registry"* sub-menu item from the *"Virtual Infrastructure"* menu item
Process - Main Success Scenario	1. Actor clicks on *"template registry"* sub-menu item from the *"Virtual Infrastructure"* 2. Actor fills the registry url 3. Actor provides the username and password to access the registry 4. Actor confirms the action by clicking on "Save" 5. The image registry is associated with the system 6. A success message is presented

Exceptions	4. (a) The actor does not click on *"Save"* i. The use case ends 5. (a) The registry is not accessible i. The system presents an error message *"Could not connect to registry"* ii. The exception is logged (b) The actor has provided a wrong username or password i. The system presents an error message *"Could not connect to registry, check the authentication details"* ii. The exception is logged (c) SDN controller/API is not available i. The system displays an error message stating *"Could not contact the controller/API"*

Table B.31: Use case: CM_FR5: Disassociate container image registry

Use Case	CM_FR5: Disassociate container image registry
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: add a centralized repository for virtual images – **Network Tenant**: have a centralized repository from which the services images can be stored – **Security Monitor**: monitor the virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – An image registry was associated with the platform via use case CM_FR4
Minimum Guarantees	An error message is presented
Success Guarantees	Container registry is successfully disassociated from the system
Trigger	Actor clicks on *"template registry"* sub-menu item from the *"Virtual Infrastructure"* menu item

Process - Main Success Scenario	1. Actor clicks on *"template registry"* sub-menu item from the *"Virtual Infrastructure"* 2. The system presents the actor with the details of the current associated registry 3. The actor clicks in *"Remove registry"* and the system presents the actor with a confirmation modal 4. Actor confirms the action by clicking on "Yes" 5. The image registry is disassociated from the system 6. A success message is presented
Exceptions	4. (a) The actor clicks on *"No"* i. The use case ends 5. (a) An exception is thrown while removing the registry i. An error message is shown containing the exception ii. The exception is logged (b) SDN controller/API is not available i. The system displays an error message stating *"Could not contact the controller/API"*

Table B.32: Use case: CM_FR6: Add container image to registry

Use Case	CM_FR5: Disassociate container image registry
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Network Tenant**: make use of the virtualized services – **Security Monitor**: monitor the virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – An image registry was associated with the platform via use case CM_FR4 – The actor has fulfilled use case CM_FR8
Minimum Guarantees	An error message is presented
Success Guarantees	Container image is added to the registry
Trigger	The actor clicks on "Add a new image" and is presented with a modal

Process - Main Success Scenario	1. The actor clicks on "Add a new image" and is presented with a modal 2. The actor fills a text input with the name for the image and the category it refers to 3. The actor selects a tarball containing the container template file from his/her own machine 4. The *tarball* is uploaded to the system 5. The actor confirms the intent by clicking "Save" 6. The image is deployed to the registry 7. A success message is displayed
Exceptions	5. (a) The actor does not click on *"Save"* i. The use case ends (b) The actor has not provided a name for the image or a *tarball* containing the template and/or a category for the template image i. An error message is shown: *"The required parameter X is missing"* ii. The use case ends 6. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) There is a syntax error in the template or there is an error building the image on the physical host i. The image is not deployed to the registry ii. An error message is shown containing the rational for the exception iii. The exception is logged

Table B.33: Use case: CM_FR7: Remove container image from registry

Use Case	**CM_FR7: Remove container image from registry**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Network Tenant**: make use of the virtualized services – **Security Monitor**: monitor the virtualized services

Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – An image registry was associated with the platform via use case CM_FR4 – A container image was deployed to the registry via use case CM_FR6 – The actor has fulfilled use case CM_FR8
Minimum Guarantees	An error message is presented
Success Guarantees	Container image is removed from the registry
Trigger	Actor finds the image to remove from the displayed list
Process - Main Success Scenario	1. The actor finds the image to remove from the displayed list 2. The actor clicks on remove image and a confirmation modal is opened 3. The actor confirms the intent by clicking on *"Yes"* 4. The image is removed from the registry 5. A success message is displayed
Exceptions	3. (a) The actor clicks on *"No"* i. The use case ends 4. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) There is an error while removing the image i. The image is not removed ii. An error message is shown containing the exception iii. The exception is logged

Table B.34: Use case: CM_FR8: List Container images from registry

Use Case	**CM_FR8: List Container images from registry**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Network Tenant**: make use of the virtualized services – **Security Monitor**: monitor the virtualized services

Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – An image registry was associated with the platform via use case CM_FR4
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor is presented with the list of images currently in the image registry
Trigger	Actor clicks on *"Template Registry"* sub-menu item from the *"Virtual Infrastructure"* menu item
Process - Main Success Scenario	1. Actor clicks on *"Template Registry"* sub-menu item from the *"Virtual Infrastructure"* 2. The actor is presented with the list of images currently in the image registry
Exceptions	2. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The registry is not available i. The system presents a popup message "Could not connect to registry" and the exception is logged

Table B.35: Use case: CM_FR9: Start Container

Use Case	**CM_FR9: Start Container**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Network Tenant**: make use of the virtualized services – **Security Monitor**: monitor the virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – An image registry was associated with the platform via use case CM_FR4 – A container image was deployed to the registry via use case CM_FR6 – The actor has fulfilled use case CM_FR8
Minimum Guarantees	An error message is presented

Success Guarantees	The container is started, is attached to the SDN network and has an assigned Ip Address
Trigger	Actor finds the container image in the list
Process - Main Success Scenario	1. Actor finds the container image in the list 2. Actor clicks in "Start container" 3. The system pulls the container image from the registry (via sub-function CM_FR20) 4. The container is started by the system 5. The container id is stored by the system 6. The container is attached to the SDN network (sub-function CM_FB18) 7. The container requests an IP Address on the SDN network (sub-function CM_FB19) 8. A success message is shown to the actor
Exceptions	4. 5. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged 7. (a) The controller fails to assign an IP address to the container i. The platform cannot recognize the container as a network host ii. The container is removed by the system iii. An error message is presented to the actor iv. The exception is logged

Table B.36: Use case: CM_FR10: Stop running container

Use Case	**CM_FR10: Stop running container**
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Network Tenant**: make use of the virtualized services – **Security Monitor**: monitor the virtualized services

Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – An image registry was associated with the platform via use case CM_FR4 – A container image was deployed to the registry via use case CM_FR6 – A container was started via use-case CM_FR9 or by the system itself (if originated by one of the virtual services) – The actor has fulfilled use case CM_FR11
Minimum Guarantees	An error message is presented
Success Guarantees	The container is stopped
Trigger	Actor finds the running container in the list provided by use case CM_FR11
Process - Main Success Scenario	1. Actor finds the running container in the list provided by use case CM_FR11 2. Actor clicks in *"Stop container"* 3. The container is stopped by the system 4. The container is detached from the SDN network (sub-function CM_FR19) 5. A success message is shown to the actor
Exceptions	3. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.37: Use case: CM_FR11: List Running Containers

Use Case	**CM_FR11: List Running Containers**
Primary Actor	Network Admin
Secondary Actors	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Security Monitor**: monitor the virtualized services

Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin or Security Monitor via use case UM_FR1 – An image registry was associated with the platform via use case CM_FR4
Minimum Guarantees	An error message is presented
Success Guarantees	The list of running containers is displayed to the actor
Trigger	Actor clicks on the *"Containers"* sub-menu item of the *"Virtual infrastructure"* menu item
Process - Main Success Scenario	1. Actor clicks on the *"Containers"* sub-menu item of the *"Virtual infrastructure"* menu item 2. The list of running containers is displayed to the actor
Exceptions	2. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.38: Use case: CM_FR12: List running containers belonging to network tenant

Use Case	**CM_FR12: List running containers belonging to network tenant**
Primary Actor	Network Admin
Secondary Actors	Security Monitor, Network Tenant
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Security Monitor**: monitor the virtualized services – **Network Tenant**: monitor his/her own running virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin, Network Tenant or Security Monitor via use case UM_FR1
Minimum Guarantees	An error message is presented
Success Guarantees	The list of running containers is displayed to the actor

Trigger	Actor clicks on the *"Containers"* sub-menu item of the *"Virtual infrastructure"* menu item
Process - Main Success Scenario	1. Actor clicks on the *"Containers"* sub-menu item of the *"Virtual infrastructure"* menu item 2. The list of running containers is displayed to the actor: (a) The list shows only the containers belonging to the tenant if the actor is a network tenant (b) The list shows all the running containers and the tenant they are associated with if the actor is either a network admin or a security monitor (c) The list can be filtered by specific tenant if the actor is a network admin or a security monitor
Exceptions	2. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.39: Use case: CM_FR13: Filter by network service name

Use Case	**CM_FR13: Filter by network service name**
Primary Actor	Network Admin
Secondary Actors	Security Monitor, Network Tenant
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Security Monitor**: monitor the virtualized services – **Network Tenant**: monitor his/her own running virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin, Network Tenant or Security Monitor via use case UM_FR1 – The actor has fulfilled use case CM_FR11 if he/she is a network administrator or a security monitor – The actor has fulfilled use case CM_FR12 if he/she is a network tenant
Minimum Guarantees	An error message is presented
Success Guarantees	The list of running containers is displayed to the actor
Trigger	Actor selects the network service name from the dropdown list of network services

Process - Main Success Scenario	1. Actor selects the network service name from the dropdown list of network services. These network services can be: (a) vNIDS (b) vHoneyPot
Exceptions	2. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The container host is not reachable i. The system presents a popup error message *"Could not reach the container host"* ii. The exception is logged

Table B.40: Use case: CM_FR14: Filter by IP address

Use Case	CM_FR14: Filter by IP address
Primary Actor	Network Admin
Secondary Actors	Security Monitor, Network Tenant
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Security Monitor**: monitor the virtualized services – **Network Tenant**: monitor his/her own running virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin, Network Tenant or Security Monitor via use case UM_FR1 – The actor has fulfilled use case CM_FR11 if he/she is a network administrator or a security monitor – The actor has fulfilled use case CM_FR12 if he/she is a network tenant
Minimum Guarantees	An error message is presented
Success Guarantees	The system updates the list of running containers showing only the container which has the requested Ip Address (if exists)
Trigger	Actor fills the Ip address search field with the Ip address to filter

Process - Main Success Scenario	1. Actor fills the Ip address search field with the Ip address to filter 2. The system updates the list of running containers showing only the container which has the requested Ip Address (if exists)
Exceptions	2. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The container host is not reachable i. The system presents a popup error message *"Could not reach the container host"* ii. The exception is logged

Table B.41: Use case: CM_FR15: Filter by Mac Address

Use Case	**CM_FR15: Filter by Mac Address**
Primary Actor	Network Admin
Secondary Actors	Security Monitor, Network Tenant
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Security Monitor**: monitor the virtualized services – **Network Tenant**: monitor his/her own running virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin, Network Tenant or Security Monitor via use case UM_FR1 – The actor has fulfilled use case CM_FR11 if he/she is a network administrator or a security monitor – The actor has fulfilled use case CM_FR12 if he/she is a network tenant
Minimum Guarantees	An error message is presented
Success Guarantees	The system updates the list of running containers showing only the container which has the requested mac address (if exists)
Trigger	Actor fills the mac address search field with the mac address to filter

Process - Main Success Scenario	1. Actor fills the mac address search field with the mac address to filter 2. The system updates the list of running containers showing only the container which has the requested mac address (if exists)
Exceptions	2. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The container host is not reachable i. The system presents a popup error message *"Could not reach the container host"* ii. The exception is logged

Table B.42: Use case: CM_FR16: Filter by network service image

Use Case	**CM_FR16: Filter by network service image**
Primary Actor	Network Admin
Secondary Actors	Security Monitor, Network Tenant
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: manage the virtualized services – **Security Monitor**: monitor the virtualized services – **Network Tenant**: monitor his/her own running virtualized services
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin, Network Tenant or Security Monitor via use case UM_FR1 – The actor has fulfilled use case CM_FR11 if he/she is a network administrator or a security monitor – The actor has fulfilled use case CM_FR12 if he/she is a network tenant
Minimum Guarantees	An error message is presented
Success Guarantees	The system updates the list of running images showing only the ones with the corresponding template image
Trigger	Actor fills the search input field
Process - Main Success Scenario	1. Actor fills the search input field 2. The system updates the list of running images showing only the ones with the corresponding template image

Exceptions	2. (a) The SDN controller/ API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The container host is not reachable i. The system presents a popup error message *"Could not reach the container host"* ii. The exception is logged

B.1.4 vNIDS package

Table B.43: Use case: VN_FR1: Enable vNIDS service

Use Case	**VN_FR1: Enable vNIDS service**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in having a global vNIDS service or in managing network tenant vNIDS services – **Network Tenant**: interest in having a vNIDS service on his own sub-network
Pre-Conditions	– The website is available – Actor must be authenticated as Network Tenant or Network Admin via use case UM_FR1 – A vNIDS container image was added to the system by a network admin via use case CM_FR6 – Actor has completed use case VN_FR21
Minimum Guarantees	An error message is shown
Success Guarantees	The vNIDS service is enabled
Trigger	The actor finds the vNIDS service in the list of available vNIDS services

Process - Main Success Scenario	1. The actor finds the vNIDS service in the list of available vNIDS services 2. The actor clicks on *"Enable vNIDS service"* 3. The system enables the vNIDS service for the specific actor by starting a vNIDS container (sub-function VN_FR12). Note this sub-function is an extension of the use case CM_FR9 since it is performed by the system itself and not by a system actor 4. A success message is displayed
Exceptions	3. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The container service is not available i. The system presents a popup message *"Could not launch container, please try again later"* and the exception is logged

Table B.44: Use case: VN_FR2: Disable vNIDS service

Use Case	**VN_FR2: Disable vNIDS service**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in having control over a global vNIDS service or network tenant vNIDS services – **Network Tenant**: interest disabling network services on his own sub-network
Pre-Conditions	– The website is available – Actor must be authenticated as Network Tenant or Network Admin via use case UM_FR1 – The vNIDS service was enabled by the network tenant via use case VN_FR1 – Actor has completed use case VN_FR21
Minimum Guarantees	An error message is shown
Success Guarantees	The vNIDS service is disabled
Trigger	The actor finds the vNIDS service in the list of available vNIDS services

Process - Main Success Scenario	1. Actor finds the vNIDS service he wants to disable in the list of available vNIDS services 2. The actor clicks on *"Disable vNIDS service"* and a confirmation modal is opened. 3. The actor confirms his intention by clicking on *"Yes"* 4. The system stops the vNIDS containers (sub-function VN_FR18) 5. The system uninstalls all the rules that were previously installed to copy the network traffic coming from the network hosts (sub-function VN_FR19) 6. The vNIDS service is disabled 7. A success message is displayed
Exceptions	3. 4. 5. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged 3. (a) The actor does not confirm the action i. The use case ends 4. (a) The container service is not available i. The system presents a popup message *"Could not stop container, please try again later"* and the exception is logged 5. (a) A network device is not available at the moment i. The system stores the removal intention and deletes the rules when the device comes back online

Table B.45: Use case: VN_FR3: Add host to vNIDS service

Use Case	**VN_FR3: Add host to vNIDS service**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in having a global vNIDS service or in managing network tenant vNIDS services – **Network Tenant**: interest in having control over the hosts being monitored by the vNIDS service on his own sub-network

Pre-Conditions	– The website is available – Actor must be authenticated as Network Tenant or Network Admin via use case UM_FR1 – The vNIDS service was enabled by the network tenant via use case VN_FR1
Minimum Guarantees	An error message is shown
Success Guarantees	The host is added to the vNIDS service
Trigger	The actor clicks in *"Add host to vNIDS service"*
Process - Main Success Scenario	1. The actor clicks in *"Add host to vNIDS service""*. This can be achieved through: (a) The network topology view (use case NM_FR3) by clicking on a host (b) From the hosts list (use case NM_FR14) 2. The actor selects the correspondent vNIDS service 3. The system starts to monitor the host traffic (sub-function VN_FR13). To accomplish this: (a) The system finds the path between the host to be monitored and the vNIDS container (sub-function VN_FR14) (b) The system installs rules to copy the traffic coming from the host or with the host destination on all devices in the path between both of them (sub-function VN_FR17) 4. A success message is displayed
Exceptions	2. (a) The actor does not select any of the available vNIDS services i. The use case ends 2. 3. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The vNIDS containers are not available i. The system launches new containers and adapts the rules (c) A failure occurs in a network device i. The system finds an alternative path and adapts the rules ii. The system launches a job to monitor the availability of the switch that stopped working so that when it comes back online the rules are adapted again

Table B.46: Use case: VN_FR4: Remove host from vNIDS Service

Use Case	**VN_FR4: Remove host from vNIDS Service**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in having a global vNIDS service or in managing network tenant vNIDS services – **Network Tenant**: interest in having control over the hosts being monitored by the vNIDS service on his own sub-network
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant or as a network admin via use case UM_FR1 – The host was added to the vNIDS service via use case VN_FR3
Minimum Guarantees	An error message is shown
Success Guarantees	The host is removed from the vNIDS service
Trigger	The actor clicks in *"Remove host from vNIDS service"*
Process - Main Success Scenario	1. The actor clicks in *"Remove host from vNIDS service"*. This can be achieved through: (a) The network topology view (use case NM_FR3) by clicking on a host (b) From the hosts list (use case NM_FR14) (c) From the list of hosts already being monitored on the vNIDS service 2. The system stops monitoring the host traffic (sub-function VN_FR15). To accomplish this: (a) The system finds the path between the host and the vNIDS container (sub-function VN_FR14) (b) The system removes the previous installed rules to copy the traffic (subfunction VN_FR16) 3. A success message is displayed

Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) A network device (in the path between host and container) stops working i. The system launches a job to monitor the availability of the switch that stopped working so that when it comes back online the rules are removed

Table B.47: Use case: VN_FR5: List hosts on vNIDS service

Use Case	**VN_FR5: List hosts on vNIDS service**
Primary Actor	Network Tenant
Secondary Actors	Network Admin, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in having a global vNIDS service or in managing network tenant vNIDS services – **Network Tenant**: interest in having control over the hosts being monitored by the vNIDS service on his own sub-network – **Security Montior**: interest in having knowledge about the hosts that are being monitored for security monitoring purposes
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant, as a security monitor or as a network admin via use case UM_FR1 – A vNIDS service was enabled via use case VN_FR1 – The actor has selected a specific vNIDS service via use case VN_FR21
Minimum Guarantees	An error message is shown
Success Guarantees	A list of hosts being monitored is displayed to the actor
Trigger	The actor has selected a specific vNIDS service via use case VN_FR21
Process - Main Success Scenario	1. The actor has selected a specific vNIDS service via use case VN_FR21 2. The actor sees the list of hosts being monitored

Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.48: Use case: VN_FR6: List all vNIDS containers

Use Case	**VN_FR6: List all vNIDS containers**
Primary Actor	Network Admin
Secondary Actors	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing the virtual infrastructure – **Security Montior**: interest in monitoring the virtual infrastructure
Pre-Conditions	– The website is available – The actor is authenticated as a security monitor or as a network admin via use case UM_FR1 – The actor has completed use case VN_FR21
Minimum Guarantees	An error message is shown
Success Guarantees	A list of running containers is displayed
Trigger	The actor clicks on the *"Containers"* link associated with a specific vNIDS service displayed by use case VN_FR21
Process - Main Success Scenario	1. The actor clicks on the *"Containers"* link associated with a specific vNIDS service displayed by use case VN_FR21 2. The actor sees the list containers associated with his vNIDS service
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) A container associated with the service is not running i. The system provides visual feedback in the list item corresponding to the failing container

Table B.49: Use case: VN_FR7: List vNIDS containers belonging to tenant

Use Case	VN_FR7: List vNIDS containers belonging to tenant
Primary Actor	Network Tenant
Secondary Actors	Network Admin, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing the virtual infrastructure – **Security Montior**: interest in monitoring the virtual infrastructure – **Network Tenant**: interest in managing his own virtual services
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant, security monitor or as a network admin via use case UM_FR1
Minimum Guarantees	An error message is shown
Success Guarantees	A list of running containers is displayed
Trigger	The actor
Process - Main Success Scenario	1. The actor clicks on the *"Containers"* link associated with a specific vNIDS service displayed by use case VN_FR21 2. The actor sees the list containers associated with his vNIDS service
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) A container associated with the service is not running i. The system provides visual feedback in the list item corresponding to the failing container

Table B.50: Use case: VN_FR8: Configure vNIDS service

Use Case	VN_FR8: Configure vNIDS service
Primary Actor	Network Tenant
Secondary Actors	Network Admin

Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in configuring all the vNIDS services – **Network Tenant**: interest in managing his own vNIDS service
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant or as a network admin via use case UM_FR1 – The vNIDS service was enabled via use case VN_FR1 – The actor has selected a vNIDS service from the list provided by use case VN_FR21
Minimum Guarantees	The actor is presented with a list of options
Success Guarantees	The actor is presented with a list of options
Trigger	The actor clicks on the *"Configure service"* in the list of vNIDS services
Process - Main Success Scenario	1. The actor clicks on the *"Configure service"* in the list of vNIDS services 2. The actor is presented with a list of options: (a) List scalability policies (via use case VN_FR10) (b) Create scalability policy (via use case VN_FR9) (c) Remove scalability policy (via use case VN_FR11)
Exceptions	

Table B.51: Use case: VN_FR9: Create Scalability Policy

Use Case	**VN_FR9: Create Scalability Policy**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in configuring all the vNIDS services – **Network Tenant**: interest in managing his own vNIDS service
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant or as a network admin via use case UM_FR1 – The use case VN_FR8 was completed

Minimum Guarantees	An error message is displayed
Success Guarantees	A scalability policy is created
Trigger	The actor clicks on *"Create scalability policy"*
Process - Main Success Scenario	1. The actor clicks on *"Create scalability policy"* 2. The actor selects the type of policy to create. This can be: (a) A scalability policy based on the container network bandwith (b) A scalability policy based on the container memory consumption (c) A scalability policy based on the container CPU usage 3. The actor fills the threshold to trigger the scalability policy 4. The actor confirms the intention by clicking on *"Save"* 5. The system launches a job to monitor the vNIDS containers based on the scalability policy (sub-function VN_FR20). Anytime the threshold is reached and depending on the type of policy: (a) The system may launch a new container to attenuate the overall load and redirect new traffic to the new container (b) The system might move the container to a different physical host (c) The system might modify existing rules to move network traffic to another container (d) The opposite will happen if the service has been scaled up and is in the state it can be scaled down again 6. A success message is displayed
Exceptions	5. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The container physical host is not available i. The system will use other physical host. If none available, the system will present an error message (c) A network device that have rules to forward traffic to a vNIDS container stops working i. The system tries to find another path to continue to provide the vNIDS container with a copy of the traffic ii. If the network device is attached to the vNIDS container, a new container is launched, the rules are modified accordingly to reflect the new path and the old container is removed

Table B.52: Use case: VN_FR10: List scalability policies

Use Case	**VN_FR10: List scalability policies**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in configuring all the vNIDS services – **Network Tenant**: interest in managing his own vNIDS service
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant or as a network admin via use case UM_FR1 – The use case VN_FR8 was completed
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor views the list of scalability policies
Trigger	Actor clicks on *"Scalability policies"*
Process - Main Success Scenario	1. Actor clicks on *"Scalability policies"* 2. The system displays the list of defined scalability policies for the service instance
Exceptions	5. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.53: Use case: VN_FR11: Remove scalability policy

Use Case	**VN_FR11: Remove scalability policy**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in configuring all the vNIDS services – **Network Tenant**: interest in managing his own vNIDS service

Pre-Conditions	– The website is available – The actor is authenticated as a network tenant or as a network admin via use case UM_FR1 – The use case VN_FR10 was completed
Minimum Guarantees	An error message is displayed
Success Guarantees	The scalability policy is removed
Trigger	Actor selects the scalability rule to be removed from the list
Process - Main Success Scenario	1. Actor selects the scalability rule to be removed from the list 2. Actor clicks on "Remove" and a modal is opened 3. Actor confirms the intention by clicking on *"Yes"* 4. The scalability rule is removed 5. The monitoring service that was launched when the scalability policy was created is removed (sub-function VN_FR22) 6. A success message is displayed
Exceptions	2. 3. (a) The actor does not confirm the action i. The use case ends 4. 5. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.54: Use case: VN_FR21: List all vNIDS services

Use Case	**VN_FR21: List all vNIDS services**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in configuring all the vNIDS services – **Network Tenant**: interest in managing his own vNIDS service
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant or as a network admin via use case UM_FR1
Minimum Guarantees	An error message is displayed

Success Guarantees	The actor views the list of available vNIDS services
Trigger	Actor clicks on the *"vNIDS"* sub-menu item of the *"Services"* menu item
Process - Main Success Scenario	1. Actor clicks on the *"vNIDS"* sub-menu item of the *"Services"* menu item 2. The system presents the actor with the list of available vNIDS services
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

B.1.5 vHoneyPot package

Table B.55: Use case: VH_FR1: Deploy vHoneypot

Use Case	**VH_FR1: Deploy vHoneypot**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing all the virtual services on the network – **Network Tenant**: interest in setting a virtual honeypot in his sub-network
Pre-Conditions	– The website is available – Actor must be authenticated as Network Tenant or Network Admin via use case UM_FR1 – The actor is in the view provided by the completion of use case VH_FR3 (if the actor is a network admin) or VH_FR4 (if the actor is a network tenant) – A vHoneypot container image was associated to the platform by a network administrator via use case CM_FR6
Minimum Guarantees	An error message is shown
Success Guarantees	The vHoneypot is deployed
Trigger	Actor clicks on *"Deploy vHoneypot"*

Process - Main Success Scenario	1. Actor clicks on *"Deploy vHoneypot"* 2. The system provides the actor with a view in which he can provide a list of Ip Addresses for the virtual honeypot service 3. The actor fills the list with the desired range of Ip addresses for the vHoneypot operation (sub-function VH_FR11) 4. The actor confirms the intent by clicking on save 5. The system redirects the network traffic with source or destination in the given Ip address range to the vHoneypot container (sub-function VH_FR8). To do this: (a) It computes the shortest network paths between all devices and the vHoneypot container (sub-function VH_FR7) (b) It installs flow rules to forward the traffic to the virtual honeypot on all previous detected network devices (sub-function VH_FR9) 6. The vHoneypot is deployed 7. A success message is displayed
Exceptions	3. (a) The list contains Ip Addresses already in use i. The system provides an error message "The specified Ip Address range contains Ip addresses already in use" and the exception is logged 4. (a) The actor does not confirm the intent by clicking on save i. The use case ends and no vHoneypot is deployed 5. 6. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.56: Use case: VH_FR2: Remove vHoneypot

Use Case	**VH_FR2: Remove vHoneypot**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing all the virtual services on the network – **Network Tenant**: interest in setting a virtual honeypot in his sub-network

Pre-Conditions	– The website is available – Actor must be authenticated as Network Tenant or Network Admin via use case UM_FR1 – The actor is in the view provided by the completion of use case VH_FR3 (if the actor is a network admin) or VH_FR4 (if the actor is a network tenant) – A vHoneypot container was deployed via use case VH_FR1
Minimum Guarantees	An error message is shown
Success Guarantees	The vHoneypot container is removed
Trigger	Actor finds the vHoneypot he wants to remove in the list of vHoneypots
Process - Main Success Scenario	1. Actor finds the vHoneypot he wants to remove in the list of vHoneypots 2. Actor clicks on *"Remove vHoneypot"* and a confirmation modal is opened 3. The actor confirms the intent by clicking on *"Yes"* 4. The vHoneypot is removed. To do this, the system: (a) Stops the vHoneypot container (sub-function VH_FR5) (b) It computes the paths between all the devices and the vHoneypot to be removed (sub-function VH_FR7) (c) It removes the forwarding rules from all the devices (sub-function VH_FR6) 5. The vHoneypot is removed 6. A success message is displayed
Exceptions	3. (a) The actor does not confirm the intent by clicking on "No" i. The use case ends and no vHoneypot is removed 4. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) The container system fails during the removal of the container i. The system presents an error message *"Could not connect to the container system, please try again later"* and the exception is logged

Table B.57: Use case: VH_FR3: List all vHoneypots

Use Case	VH_FR3: List all vHoneypots
Primary Actor	Network Admin
Secondary Actors	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing all the virtual services on the network – **Security Monitor**: interest in monitoring all the virtual services on the network
Pre-Conditions	– The website is available – Actor must be authenticated as a Security Monitor or as a Network Admin via use case UM_FR1
Minimum Guarantees	An error message is shown
Success Guarantees	The actor sees the list of previously deployed vHoneypots
Trigger	Actor clicks in the vHoneypot sub-menu item of the *"Services"* menu item
Process - Main Success Scenario	1. Actor clicks in the vHoneypot sub-menu item of the *"Services"* menu item 2. The system presents the list of all the previously deployed vHoneypots
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.58: Use case: VH_FR4: List vHoneypots belonging to Tenant

Use Case	VH_FR4: List vHoneypots belonging to Tenant
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal

Stakeholders and Interests	– **Network Admin**: interest in managing the virtual services on his sub-network – **Network Tenant**: interest in monitoring all the virtual services on the network
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Tenant or as a Network Admin via use case UM_FR1 – The actor is a network admin and has completed use case VH_FR3
Minimum Guarantees	An error message is shown
Success Guarantees	The actor sees the list of previously deployed vHoneypots
Trigger	Actor clicks in the vHoneypot sub-menu item of the *"Services"* menu item
Process - Main Success Scenario	1. Actor clicks on the *"vHoneypot"* sub-menu item of the *"Services"* main menu item. If the actor is a network admin: (a) The actor selects the Tenant name from the Network Tenants filtering dropdown list 2. The system presents the list of all the previously deployed vHoneypots
Exceptions	1. 2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

B.1.6 Data_Diode package

Table B.59: Use case: DD_FR1: Set Network Link as a data diode

Use Case	DD_FR1: Set Network Link as a data diode
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing all the virtual services on the network – **Network Tenant**: interest in setting a network link as a data diode

Pre-Conditions	– The website is available – Actor must be authenticated as Network Tenant or Network Admin via use case UM_FR1 – The actor has fulfilled use case NM_FR15 or NM_FR11 (if a network admin) or NM_FR12 (if a network tenant or network admin)
Minimum Guarantees	An error message is shown
Success Guarantees	The link is set as a data diode in the specified direction
Trigger	Actor clicks on *"Set link as a data diode"*
Process - Main Success Scenario	1. Actor clicks on *"Set link as a data diode"*. This can achieved by: (a) Clicking on the button on the modal that opens after clicking on a link in the network topology view (if the actor is a network admin and has fulfilled use case NM_FR11 or if the actor is a network tenant and has fulfilled use case NM_FR12 or NM_FR11) (b) Clicking directly on the button in the list of network links (if the actor has fulfilled use case NM_FR15) 2. The system presents the actor the the two directions of the network link 3. The actor selects one of the directions 4. The actor confirms his intent by clicking on *"Save"* 5. The system install rules to drop the network packages on the network device that contains the edge link (sub-function DD_FR9) 6. The link is set as a data diode in the specified direction 7. A success message is shown

Exceptions	1. (a) The link selected is not an edge link i. The use case ends. The actor has to click on an edge link 2. (a) The link is already set as a data diode on one of the link directions i. The system also shows the two directions but identifies correctly the direction the data diode is already set. The direction defined as a data diode has no option to "set direction as a data diode" 3. (a) The actor does not select any direction i. The use case ends 4. (a) Actor does not click on save i. The use case ends 5. 6. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.60: Use case: DD_FR2: Set Network Link as a regular link

Use Case	**DD_FR2: Set Network Link as a regular link**
Primary Actor	Network Tenant
Secondary Actors	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing all the virtual services on the network – **Network Tenant**: interest in setting a network link as a data diode
Pre-Conditions	– The website is available – Actor must be authenticated as Network Tenant or Network Admin via use case UM_FR1 – The actor has fulfilled use case DD_FR3 (if the actor is a network tenant or network admin) or DD_FR4 (if the actor is a network admin) – A link was previously set as a data diode via use case DD_FR1
Minimum Guarantees	An error message is shown
Success Guarantees	The link is set as a regular link

Trigger	Actor clicks on *"remove data diode"*
Process - Main Success Scenario	1. Actor clicks on *"remove data diode"*. This can achieved by: (a) Clicking on the button on the modal that opens after clicking on a link previously set as a data diode in the network topology view (if the actor is a network admin and has fulfilled use case NM_FR11 or if the actor is a network tenant and has fulfilled use case NM_FR12 or NM_FR11) (b) Clicking directly on the button in the list of network links set as data diode (via use case DD_FR3 if the actor is a network tenant or via use case DD_FR4 if the actor is a network admin) 2. The system presents the actor with a confirmation modal 3. The actor confirms his intent by clicking on "Yes" 4. The system remove the rules to drop the network packages on the network device that contains the edge link (sub-function DD_FR7) 5. The link is set as a regular link 6. A success message is shown
Exceptions	3. (a) Actor does not click on *"Yes"* i. The use case ends 2. (a) The link is already set as a data diode on one of the link directions i. The system also shows the two directions but identifies correctly the direction the data diode is already set. The direction defined as a data diode has no option to "set direction as a data diode" 5. 6. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.61: Use case: DD_FR3: List all network links set as a data diode

Use Case	**DD_FR3: List all network links set as a data diode**
Primary Actor	Network Admin
Secondary Actors	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal

Stakeholders and Interests	– **Network Admin**: interest in managing all the virtual services on the network – **Security Monitor**: interest in monitoring all the virtual services on the network
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin or Security Monitor via use case UM_FR1
Minimum Guarantees	An error message is shown
Success Guarantees	The system presents the actor with the list of network links set as a data diode
Trigger	Actor clicks on the *"Data Diode"* sub-menu item of the "Services" menu item
Process - Main Success Scenario	1. Actor clicks on the *"Data Diode"* sub-menu item of the "Services" menu item 2. The system presents the actor with the list of network links set as a data diode. From there the actor can: (a) Filter the data diodes by network tenant via use case DD_FR6 (b) Filter the data diodes by sub-network name via use case DD_FR7 (c) Perform any other filters that are available for use case NM_FR15
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.62: Use case: DD_FR4: Filter network link by sub-network name

Use Case	**DD_FR4: Filter network link by sub-network name**
Primary Actor	Network Admin
Secondary Actors	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing all the virtual services on the network – **Security Monitor**: interest in monitoring all the virtual services on the network

Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin or Security Monitor via use case UM_FR1 – The actor has to have fulfilled use case DD_FR6
Minimum Guarantees	An error message is displayed
Success Guarantees	The system presents an updated list of network links
Trigger	The actor fills the search field with the sub-network name
Process - Main Success Scenario	1. The actor fills the search field with the sub-network name 2. The system presents an updated list of network links (set as a data diode)
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.63: Use case: DD_FR5: Filter network link by network tenant

Use Case	DD_FR5: Filter network link by network tenant
Primary Actor	Network Admin
Secondary Actors	Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Admin**: interest in managing all the virtual services on the network – **Security Monitor**: interest in monitoring all the virtual services on the network
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin or Security Monitor via use case UM_FR1 – The actor has to have fulfilled use case DD_FR6
Minimum Guarantees	An error message is displayed
Success Guarantees	The system presents an updated list of network links
Trigger	The actor fills the search field with the tenant name

Process - Main Success Scenario	1. The actor fills the search field with the tenant name 2. The system presents an updated list of network links (set as a data diode)
Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.64: Use case: DD_FR6: List all network links set as a data diode for links belonging to a tenant sub-network

Use Case	**DD_FR6: List all network links set as a data diode for links belonging to a tenant sub-network**
Primary Actor	Network Tenant
Secondary Actors	Network Admin, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in managing his virtual data diode service – **Network Admin**: interest in managing all the virtual services on the network – **Security Monitor**: interest in monitoring all the virtual services on the network
Pre-Conditions	– The website is available – Actor must be authenticated as a Network Tenant, Network Admin or Security Monitor via use case UM_FR1 – In the case the actor is a network admin or a security monitor he has to have fulfilled use case DD_FR6 previously
Minimum Guarantees	An error message is displayed
Success Guarantees	The system presents the actor with the list of network links set as a data diode
Trigger	Actor clicks on the "Data Diode" sub-menu item of the "Virtual infrastructure" menu item if he is a network tenant. Actor finishes DD_FR6 otherwise.
Process - Main Success Scenario	1. Actor clicks on the "Data Diode" sub-menu item of the "Virtual infrastructure" menu item if he is a network tenant. Actor finishes DD_FR6 otherwise. 2. The system presents the actor with the list of network links set as a data diode

Exceptions	2. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

B.1.7 Network_Event_Factory package

Table B.65: Use case: NEF_FR1: Add message broker topic URI

Use Case	NEF_FR1: Add message broker topic URI
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: be able to have the system to publish network events to an external SIEM – **Security Admin**: management of SDN events – **Security Monitor**: monitoring the SDN network – **ystem Admin**: management of SDN events
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1
Minimum Guarantees	An error message is presented
Success Guarantees	The broker and topic URL are added to the system
Trigger	Actor clicks on *"Network Event factory"* menu item
Process - Main Success Scenario	1. Actor clicks on *"Network Event factory"* menu item 2. The actor fills the broker and topic URL in the view presented by the system 3. The system connects to the provided URL and topic 4. The actor selects the kind of events he wants the platform to publish to an external system. This consists of a list of checkboxes: (a) Device Events (b) Link Events (c) Topology Events (d) Host Events (e) Controller Events 5. The actor confirms his selection by clicking on "Save" 6. The system presents a success message 7. The system starts to publish events to the broker topic via sub-function NEF_FR10

Exceptions	2. (a) The broker or the topic do not exist i. The system presents a popup message *"Could not contact broker"* and the exception is logged ii. The system presents a popup message *"Topic does not exist"* and the exception is logged 5. (a) The actor does not click on *"save"* i. The use case ends (b) The SDN/Controller API is not available i. The system shows a popup stating *"Could not contact backend please try again later"* and the exception is logged 6. (a) An exception occurs while adding the broker URI i. A popup is shown containing information about the exception and the exception is also logged

Table B.66: Use case: NEF_FR2: Remove message broker topic URI

Use Case	NEF_FR2: Remove message broker topic URI
Primary Actor	Network Admin
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: be able to have full control over whether or not the SDN sub-system publishes events to an external SIEM
Pre-Conditions	– The website is available – Actor must be authenticated as Network Admin via use case UM_FR1 – The broker and topic uri were added to the platform via use case NEF_FR2
Minimum Guarantees	An error message is presented
Success Guarantees	The broker and topic URI are removed from the system
Trigger	Actor clicks on *"Network Event factory"* menu item
Process - Main Success Scenario	1. Actor clicks on *"Network Event factory"* menu item 2. The actor clicks on a remove button located near the broker and topic (SIEM) settings and a confirmation modal is opened 3. The actor confirms his intent by clicking on *"Yes"* 4. The system presents a success message 5. The system stops publishing events to the broker topic

Exceptions	3. (a) The actor does not click on "Yes" i. The use case ends without changes to the system 5. (a) The SDN/Controller API is not available i. The system shows a popup stating *"Could not contact backend please try again later"* and the exception is logged (b) An exception is thrown while removing the URI from the system i. An error message is shown containing the exception information ii. The exception is logged

Table B.67: Use case: NEF_FR3: Publish network events

Use Case	NEF_FR3: Publish network events
Primary Actor	-
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: be able to have the system to publish network events to an external SIEM – **Security Admin**: management of SDN events – **Security Monitor**: monitoring the SDN network – **System Admin**: management of SDN events
Pre-Conditions	– Use case NEF_FR1 was completed – At least an SDN controller node is running
Minimum Guarantees	An exception is logged
Success Guarantees	The network event is published to the broker and topic previously configured via use case NEF_FR1
Trigger	A network event occurs in the SDN network
Process - Main Success Scenario	1. A network event occurs in the SDN network 2. The system publishes the event to the broker and topic defined via use case NEF_FR1. The event can be published: (a) Via use case NEF_FR4 if it is a device event (b) Via use case NEF_FR5 if it is a link event (c) Via use case NEF_FR6 if it is a topology event (d) Via use case NEF_FR7 if it is a host event (e) Via use case NEF_FR8 if it is a controller event
Exceptions	2. (a) An exception is thrown during an event i. The exception is logged

Table B.68: Use case: NEF_FR4: Publish device events

Use Case	**NEF_FR4: Publish device events**
Primary Actor	-
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: be able to have the system to publish network events to an external SIEM – **Security Admin**: management of SDN events – **Security Monitor**: monitoring the SDN network – **System Admin**: management of SDN events
Pre-Conditions	– Use case NEF_FR3
Minimum Guarantees	An exception is logged
Success Guarantees	The device event is published to the broker and topic previously configured via use case NEF_FR1
Trigger	A network event occurs in the SDN network
Process - Main Success Scenario	1. A device event occurs in the SDN network 2. The system publishes the event to the broker and topic defined via use case NEF_FR1. The event can be of type: (a) DEVICE_ADDED (b) DEVICE_AVAILABILITY_CHANGED (c) DEVICE_REMOVED (d) DEVICE_SUSPENDED (e) DEVICE_UPDATED (f) PORT_ADDED (g) PORT_REMOVED (h) PORT_STATS_UPDATED (i) PORT_UPDATED
Exceptions	2. (a) An exception is thrown during an event i. The exception is logged

Table B.69: Use case: NEF_FR5: Publish link events

Use Case	**NEF_FR5: Publish link events**
Primary Actor	-
Scope	SDN (Sub-system) - Black-box
Level	System-Goal

Stakeholders and Interests	– **Network Admin**: be able to have the system to publish network events to an external SIEM – **Security Admin**: management of SDN events – **Security Monitor**: monitoring the SDN network – **System Admin**: management of SDN events
Pre-Conditions	– Use case NEF_FR3
Minimum Guarantees	An exception is logged
Success Guarantees	The link event is published to the broker and topic previously configured via use case NEF_FR1
Trigger	A network event occurs in the SDN network
Process - Main Success Scenario	1. A link event occurs in the SDN network 2. The system publishes the event to the broker and topic defined via use case NEF_FR1. The event can be of type: (a) LINK_ADDED (b) LINK_REMOVED (c) LINK_UPDATED
Exceptions	2. (a) An exception is thrown during an event i. The exception is logged

Table B.70: Use case: NEF_FR6: Publish topology events

Use Case	NEF_FR6: Publish topology events
Primary Actor	-
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: be able to have the system to publish network events to an external SIEM – **Security Admin**: management of SDN events – **Security Monitor**: monitoring the SDN network – **System Admin**: management of SDN events
Pre-Conditions	– Use case NEF_FR3
Minimum Guarantees	An exception is logged
Success Guarantees	The topology event is published to the broker and topic previously configured via use case NEF_FR1
Trigger	A topology event occurs in the SDN network

Process - Main Success Scenario	1. A topology event occurs in the SDN network 2. The system publishes the event to the broker and topic defined via use case NEF_FR1. The event can be of type: (a) TOPOLOGY_CHANGED
Exceptions	2. (a) An exception is thrown during an event i. The exception is logged

Table B.71: Use case: NEF_FR7: Publish host events

Use Case	**NEF_FR7: Publish host events**
Primary Actor	-
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: be able to have the system to publish network events to an external SIEM – **Security Admin**: management of SDN events – **Security Monitor**: monitoring the SDN network – **System Admin**: management of SDN events
Pre-Conditions	– Use case NEF_FR3
Minimum Guarantees	An exception is logged
Success Guarantees	The host event is published to the broker and topic previously configured via use case NEF_FR1
Trigger	An host event occurs in the SDN network
Process - Main Success Scenario	1. An host event occurs in the SDN network 2. The system publishes the event to the broker and topic defined via use case NEF_FR1. The event can be of type: (a) HOST_ADDED (b) HOST_MOVED (c) HOST_REMOVED (d) HOST_UPDATED
Exceptions	2. (a) An exception is thrown during an event i. The exception is logged

Table B.72: Use case: NEF_FR8: Publish network controller events

Use Case	NEF_FR8: Publish network controller events
Primary Actor	-
Scope	SDN (Sub-system) - Black-box
Level	System-Goal
Stakeholders and Interests	– **Network Admin**: be able to have the system to publish network events to an external SIEM – **Security Admin**: management of SDN events – **Security Monitor**: monitoring the SDN network – **System Admin**: management of SDN events
Pre-Conditions	– Use case NEF_FR3
Minimum Guarantees	An exception is logged
Success Guarantees	The network controller event is published to the broker and topic previously configured via use case NEF_FR1
Trigger	A network controller event occurs in the SDN network
Process - Main Success Scenario	1. A network controller event occurs in the SDN network 2. The system publishes the event to the broker and topic defined via use case NEF_FR1. The event can be of type: (a) APP_ACTIVATED (b) APP_DEACTIVATED (c) APP_INSTALLED (d) APP_PERMISSIONS_CHANGED (e) APP_UNINSTALLED (f) CLUSTER_INSTANCE_ACTIVATED (g) CLUSTER_INSTANCE_ADDED (h) CLUSTER_INSTANCE_DEACTIVATED (i) CLUSTER_INSTANCE_REMOVED (j) RULE_ADD_REQUESTED (k) RULE_ADDED (l) RULE_REMOVE_REQUESTED (m) RULE_REMOVED (n) RULE_UPDATED (o) INTENT_INSTALLED (p) INTENT_WITHDRAWN (q) INTENT_PURGED (r) INTENT_CORRUPT (s) INTENT_FAILED (t) MASTERSHIP_MASTER_CHANGED (u) MASTERSHIP_BACKUPS_CHANGED (v) PACKET_EMIT

Exceptions	2. (a) An exception is thrown during an event i. The exception is logged

B.2 Monitoring Package

B.2.1 Network_Statistics package

Table B.73: Use case: NS_FR1: Network Statistics

Use Case	NS_FR1: Network Statistics
Primary Actor	Network Admin
Secondary Actors	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in monitoring the actor sub-network assets – **Network Admin**: interest in monitoring the overall network – **Security Monitor**: interest in monitoring the overall network
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant, as a security monitor or as a network admin via use case UM_FR1 – The user has completed use case NM_FR10 if he is a network tenant – The user has completed use case NM_FR9 or NM_FR10 if he is a network admin or security monitor
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor can access statistical information regarding the network
Trigger	The actor clicks on the *"Statistics"* menu item
Process - Main Success Scenario	1. The actor clicks on the *"Statistics"* menu item 2. The system displays three menu items from where the user can: (a) View host statistics via use case NS_FR2 (b) View device statistics via use case NS_FR3 (c) View link statistics via use case NS_FR4
Exceptions	-

Table B.74: Use case: NS_FR2: View host statistics

Use Case	NS_FR2: View host statistics
Primary Actor	Network Admin
Secondary Actors	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in monitoring the actor sub-network assets – **Network Admin**: interest in monitoring the overall network – **Security Monitor**: interest in monitoring the overall network
Pre-Conditions	– The website is available – The user has completed use case NS_FR1
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor can access statistical information regarding host
Trigger	The actor clicks on *"Hosts"*
Process - Main Success Scenario	1. The actor clicks on *"Hosts"* 2. The actor selects the respective host from the displayed list 3. The system displays the host statistics. These include: (a) Global inbound traffic (total and realtime) (b) Global outbound traffic traffic (total and realtime) (c) Inbound traffic form other hosts (total and realtime) (d) Outbound traffic to other hosts (total and realtime) 4. The view are also displayed in the form of live charts (sub-function NS_FR5)
Exceptions	3. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.75: Use case: NS_FR3: View device statistics

Use Case	NS_FR3: View device statistics
Primary Actor	Network Admin
Secondary Actors	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box

Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in monitoring his network assets – **Network Admin**: interest in monitoring the overall network – **Security Monitor**: interest in monitoring the overall network
Pre-Conditions	– The website is available – The user has completed use case NS_FR1
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor can access statistical information regarding device
Trigger	The actor clicks on *"Devices"*
Process - Main Success Scenario	1. The actor clicks on *"Devices"* 2. The actor selects the respective host device the the provided list 3. The system displays the device statistics. These include: (a) Statistics per port (total and real time) (b) Statistics per flow (total and real time) 4. The view are also displayed in the form of live charts (sub-function NS_FR5)
Exceptions	3. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

Table B.76: Use case: NS_FR4: View link statistics

Use Case	**NS_FR4: View link statistics**
Primary Actor	Network Admin
Secondary Actors	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in monitoring his network assets – **Network Admin**: interest in monitoring the overall network – **Security Monitor**: interest in monitoring the overall network
Pre-Conditions	– The website is available – The user has completed use case NS_FR1
Minimum Guarantees	An error message is displayed

Success Guarantees	The actor can access statistical information regarding link
Trigger	The actor clicks on *"Links"*
Process - Main Success Scenario	1. The actor clicks on *"Links"* 2. The actor selects the respective link in the displayed list 3. The system displays the link statistics. These include: (a) Statistics per port (total and real time) (b) Statistics per flow (total and real time) (c) Statistics per host (inbound and outbound traffic - total e real time) The view are also displayed in the form of live charts (sub-function NS_FR5)
Exceptions	3. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged

B.2.2 Container_Statistics package

Table B.77: Use case: CS_FR1: Container real-time statistics

Use Case	CS_FR1: Container real-time statistics
Primary Actor	Network Admin
Secondary Actors	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in monitoring his network assets – **Network Admin**: interest in monitoring the overall network – **Security Monitor**: interest in monitoring the overall network
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant, as a security monitor or as a network admin via use case UM_FR1 – The user has completed use case CM_FR12 if he is a network tenant – The user has completed use case CM_FR11 or CM_FR12 if he is a network admin or security monitor
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor can access statistical information regarding the container
Trigger	The actor selects a specific container from the list of running containers

Process - Main Success Scenario	1. The actor selects a specific container from the list of running containers 2. The actor clicks on the "Statistics" option of the list item 3. The system redirects the actor to a view where the actor can: (a) View CPU usage via use case CS_FR3 (b) View network bandwidth usage via use case CS_FR4 (c) View memory usage via use case CS_FR2
Exceptions	-

Table B.78: Use case: CS_FR2: View memory consumption

Use Case	**CS_FR2: View memory consumption**
Primary Actor	Network Admin
Secondary Actors	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in monitoring his network assets – **Network Admin**: interest in monitoring the overall network – **Security Monitor**: interest in monitoring the overall network
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant, as a security monitor or as a network admin via use case UM_FR1 – The user has completed use case CS_FR1
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor can see the real-time memory usage of the container
Trigger	The actor clicks on the *"Memory"* tab
Process - Main Success Scenario	1. The actor clicks on the *"Memory"* tab 2. The system displays the memory consumption of the container. The provided information is: (a) Real time memory consumption (b) Percentage of host used memory (c) Percentage of used memory (if the container was launched with memory limits) Data is displayed in live charts (sub-function NS_FR5)

Exceptions	3. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) Container engine is not available i. The use case ends with an error message ii. The exception is logged

Table B.79: Use case: CS_FR3: View host CPU usage

Use Case	**CS_FR3: View host CPU usage**
Primary Actor	Network Admin
Secondary Actors	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in monitoring his network assets – **Network Admin**: interest in monitoring the overall network – **Security Monitor**: interest in monitoring the overall network
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant, as a security monitor or as a network admin via use case UM_FR1 – The user has completed use case CS_FR1
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor can see the real-time CPU usage of the container
Trigger	The actor clicks on the *"CPU"* tab
Process - Main Success Scenario	1. The actor clicks on the *"CPU"* tab 2. The system displays the CPU usage of the container. The provided information is: (a) Percentage of host used CPU (b) Percentage of container used CPU (if the container was launched with CPU limitations) Data is displayed in live charts (sub-function NS_FR5)
Exceptions	3. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) Container engine is not available i. The use case ends with an error message ii. The exception is logged

Table B.80: Use case: CS_FR4: View Network bandwidth usage

Use Case	CS_FR4: View Network bandwidth usage
Primary Actor	Network Admin
Secondary Actors	Network Tenant, Security Monitor
Scope	SDN (Sub-system) - Black-box
Level	User-Goal
Stakeholders and Interests	– **Network Tenant**: interest in monitoring his network assets – **Network Admin**: interest in monitoring the overall network – **Security Monitor**: interest in monitoring the overall network
Pre-Conditions	– The website is available – The actor is authenticated as a network tenant, as a security monitor or as a network admin via use case UM_FR1 – The user has completed use case CS_FR1
Minimum Guarantees	An error message is displayed
Success Guarantees	The actor can see the real-time network bandwidth used by the container
Trigger	The actor clicks on the *"Network"* tab
Process - Main Success Scenario	1. The actor clicks on the *"Network"* tab 2. The system displays the network usage of the container. This information refers to: (a) TX byte count (b) Number of TX packets (c) TX dropped packet number (d) TX dropped byte count (e) RX byte count (f) Number of RX packets (g) RX dropped packet number (h) RX dropped byte count Data is displayed in live charts (sub-function NS_FR5) Data above refers to the container management interface but it is also available for the SDN network interface (via use case NFS_FR2)
Exceptions	3. (a) SDN controller/API is not available i. The system presents a popup message *"Could not connect to the controller, please try again later"* and the exception is logged (b) Container engine is not available i. The use case ends with an error message ii. The exception is logged

Appendix C

External Interfaces

C.1 Network Management

C.1.1 HTTP endpoints

- `/sdn/networks` – Used to globally access network information

  GET – Provides a list of created logical sub-network. The result depends on the role of the user calling the method. If the user is a network tenant, the API will return only the networks that are associated with his account. Example response:

```
{
  "networks": [
  {
    "name":"datacenter",
    "hosts":["02:42:25:71:08:22/NONE"],
    "tenants":["tenant1"]
  }
  ]
}
```

  POST # – Creates a new sub-network. It accepts the following JSON payload:

```
{"name": "datacenter"}
```

- `/sdn/networks/<network>` – Manipulate a specific logical-network (where <network> refers to the logical network name)

  GET – Lists the information associated with a specific sub-network. This includes the list of Hosts belonging to the network and the tenant to which the network is associated (see the former endpoint)

  DELETE # – Removes the given network

- `/sdn/networks/<network>/hosts` – Access logical network hosts

  GET – Lists all the hosts present in the logical network

- `/sdn/networks/<network>/hosts/<hostid>` – Manipulates the association of hosts and networks

  POST – Adds the host to the network (no payload)

Admin only

DELETE – Removes the host from the network

GET – Obtain detailed information about a given host. Example response:


```
{
  "id":"00:0C:29:C6:0A:18/None",
  "ip":["192.168.1.11"],
  "mac":"00:0C:29:C6:0A:18",
  "vlan":null,
  "type":"host",
  "assetType":"HOST",
  "realtype":"host",
  "inNetworks":["datacenter"],
  "canBeAddedToNetworks":[]
}
```


- `/sdn/networks/<network>/tenants` – Refer to the network-tenant relationship

  GET # – List all the tenants that own the network (i.e. can deploy services on the network)

- `/sdn/networks/tenants/<network>` # – Links to the above endpoint

- `/sdn/networks/<network>/tenants/<tenant>` – Manipulates the association of tenants and networks (<tenant> refers to the username)

  POST – Associates the tenant with the network (no payload)

  DELETE – Dissociates the tenant from the network

- `/sdn/networks/<network>/devices` #

  GET – Lists all the devices (OpenFlow enabled switches) belonging to a subnetwork

- `/sdn/networks/<network>/devices/<deviceid>`

  GET – Obtain the properties of a given switch. Example response:


```
{
    "type":"device",
    "assetType":"SWITCH",
    "id":"of:0000a0369f4c8f7a",
    "vendor":"Nicira, Inc.",
    "HW":"Open vSwitch",
    "SW":"2.7.0",
    "Serial":"None",
    "Protocol":"OF_13",
    "Ports":4,
    "Flows":3
}
```


- `/sdn/networks/<network>/links`

  GET – Obtain the list of links in the network

- `/sdn/networks/<network>/links/<linkid>`

  GET – Obtain specific link information. Example response:

Admin only

```
{
    "type":"link",
    "Linktype":"Edge",
    "aType":"host",
    "aId":"00:0C:29:C6:0A:18/None",
    "bType":"device",
    "bId":"of:0000a0369fbc27f0",
    "bPort":3,
    "belongsToDataDiodeServices":[],
    "canBeSetAsDataDiodeInNetworks":["datacenter"]
}
```

C.1.2 Web-sockets endpoints

- `/ws/network` – Provides the topology graph as streaming websocket frames each 2 seconds. The graph of each sub-network is combined to provide a global view of the selected networks. It expects the frame below to start/or change the streaming data.

```
{
  "action":"topology",
  "networks":["datacenter", "networkx"]
}
```

In simple terms, the topology graph has a representation similar to the one presented below.

```
{
  "nodes":[
    {"id":"of:0000a0369fbc27f0",...},
    {"id":"of:0000a0369fbc27f0"...}...
  ]
  "edges":[
    {
      "from":"of:0000a0369fbc27f0",
      "to":"of:0000a0369f4c8fff", ...
    }],
  "event":"topologygraph"
}
```

C.1.3 Command line

- `add-host <network> <hostid>` - add host to a network
- `associate-user-network <network> <username>` - associate a user with a network
- `create-network <network>` - create a network
- `disassociate-user-network <network> <username>` - disassociate a user from a network
- `get-all-hosts` - list all topology hosts
- `get-hosts <network>` - list all hosts in a network
- `list-networks` - list all created networks

- `get-tenants <network>` - list all users (tenants) associated with a network
- `get-networks <username>` - list all networks belonging to a given tenant
- `network-management-clearall` - "factory reset"
- `remove-host <network> <hostid>` - remove an host from a network
- `remove-network <network>` - remove a network

C.2 Docker integration

C.2.1 HTTP endpoints

- `/sdn/docker/registry` # – Manages the private registry information

  GET – Obtain the url of the registry associated with the platform

  POST – Associate a registry with the platform. It expects the following JSON payload:

```
{
   "url": "https://registry.iads.dei.uc.pt",
   "username": "xxxx",
   "password": "xxxx"
}
```

  DELETE #– Removes the registry from the controller datastore (no payload)

- `/sdn/docker/registry/<repo>/upload` #– Manages the private registry information

  POST – Uploads a given container image to the registry and to a specific repository (e.g. vNIDS). This is a multi-part upload, the expected payload:

```
{
  "name": "snort",
  "description": "container image description",
  "file": "a reference to the uploaded file",
  "image":"a logo for the container"
}
```

- `/sdn/docker/registry/<repo>/<image>` # – Manages specific images

  GET – Obtain the information regarding an image (name, description, logo)

  DELETE – Removes the image from the repository

- `/sdn/docker/nodes` – Manages the virtualization nodes

  GET – Obtain the list of virtualization nodes associated with the platform

  POST – Associate a new node with the platform. Payload:

Admin only

```
{
  "url": "https://docker1.dei.uc.pt:4243"
}
```

- `/sdn/docker/containers` #

  GET – Obtain a list of all the containers running in the virtualization infrastructure pool. Example response:

```
{
  "containers":[
    {
    "id":"693a8jf3ldl2",
    "state": "running",
    "issdn": true,
    "image": "snort",
    "tenant": "tenant1",
    "network": "datacenter",
    "serviceId": "asdsa343sdfdf",
    "category": "vnids"
   }
  ]
}}
```

- `/sdn/docker/containers/<containerid>` #

  GET – Obtain the information of a specific container

- `/sdn/docker/containers/<containerid>/logs` #

  GET – Obtain the container runtime logs as a string

- `/sdn/docker/containers/<containerid>/top` #

  GET – List the processes running in the containers as a string

C.2.2 Web-sockets endpoints

- `/ws/docker/` # – The websocket endpoint expects the following message frames:
  - To start streaming the list of nodes and their details with a new frame containing the state of all the virtualization nodes (CPU usage, RAM usage, docker version, operating system, number of running containers) each two seconds:

```
{"action": "nodestreamstart"}
```

  - To stop streaming the list of nodes:

```
{"action":"nodestreamstop"}
```

  - To start streaming the list of containers and their state. The list always reflects the current state of each container:

```
{"action": "containerliststart"}
```

Admin only

– To stop streaming the list of containers:

```
{"action": "containerliststart"}
```

– To stream the network, CPU, RAM and SDN network statistics of a given container :

```
{"action": "containerstatsstart", "containerid": "xxx"}
```

– Stops streaming the container stats:

```
{"action": "containerstatsstop"}
```

C.2.3 Command line

- `docker-node-add <url>` - Add a docker node to the virtualization infrastructure
- `docker-registry-add <registry>` - Add a public (unauthenticated) registry to the platform
- `docker-registry-add-auth <url> <username> <password>` - Add a private registry to the platform
- `docker-get-container <containerId>` - Check for the existence of a container in the docker node pool
- `docker-containers-list` - List all containers in the docker node pool
- `docker-node-list` - List all docker nodes associated with the platform
- `docker-node-remove` - Remove a docker node from the platform
- `docker-node-list-info` - List all docker nodes with extended information
- `docker-container-start <repository> <image> <serviceId> <username> <network>` - Start a container with a given image and attach it to a user and network
- `docker-container-stats <containerId>` - List a container statistics
- `docker-container-stats-stream <containerid> <period>` - Stream container statistics for a given amount of time
- `docker-container-ashost <containerId>` - Find the hostId (ONOS topology) of a running container
- `docker-get-registry` - List the registry associated with the platform along with its information
- `docker-registry-image-details <repository> <image>` - Get the image details from the registry
- `docker-registry-list-images <repository>` - List all images for a given repository (vnids, vhoneypot) from the registry
- `docker-registry-list-image-details` - Same as above but with extended information/details
- `docker-registry-list-repositories` - List all repositories in the registry

- `docker-rm-registry` - Remove the registry from the platform
- `docker-registry-remove-image <repository> <image>` - Delete an image from the registry
- `docker-container-logs-since-now <containerid> <time>` - Stream container logs for a given amount of time

C.3 vNIDS

C.3.1 HTTP endpoints

- `/sdn/vnids` – global vNIDS namespace

  GET – Lists all the vnids services that have been instantiated. Example response:

```
{
  "vnids":[
    {
      "serviceid":"65d49a4f1dd067c46439ef096bb64a6a",
      "username":"admin",
      "network":"datacenter",
      "image":"snort",
      "hostsMonitored":
        [
          "00:00:00:00:00/None",
          "00:00:00:01:00/None"
        ],
        "containers":[
          {
            "id":"45c7f15dc3a64",
             "ip":"192.168.0.82",
             "mac":"C2:79:0A:30:0E:92",
             "hostid":"C2:79:0A:30:0E:92/None"
           }
        ]
    }
  ]
}
```

  POST – Create a new vNIDS service. Example of a POST JSON payload:

```
{
  "network":"datacenter",
  "image":"snort"
}
```

  Provided response:

```
{
  "serviceId":"65d49a4f1dd067c46439ef096bb64a6a",
}
```

- `/sdn/vnids/<serviceid>` – Manages a specific vNIDS service

  GET – Returns the information for the service (see `/sdn/vnids` endpoint)

DELETE – Remove the vNIDS service instance

- `/sdn/vnids/<serviceid>/<hostid>` – Manages host monitoring on each service

  POST – The provided host will have its traffic replicated to the vNIDS container (no payload)

  DELETE – The service will stop monitoring the traffic of the provided host

C.3.2 Command line

- `vnids-add-host <serviceid> <hostid>` - Add an host to a vNIDS service
- `create-vnids <username>` - Create a vNIDS service and link it with a tenant
- `vnids-delete-all` - "Factory reset"
- `vnids-list` - List all vNIDS serviceIds
- `vnids-list-details` - List all vNIDS with details
- `vnids-setimage <serviceid> <repository/image>` - Associate an image with the service (pre-deployment)
- `vnids-setnetwork <serviceid> <network>` - Associate a network with the service (pre-deployment)
- `vnids-remove <serviceid>` - Delete a specific service

C.4 vHoneypot

C.4.1 HTTP endpoints

- `/sdn/vhoneypot` – Global vHoneypot url namespace

  GET – Lists all the vHoneypot services that have been instantiated. Example response:


```
{
  "vhoneypots":[
    {
      "serviceid":"65d49a4f1dd067c46439ef096bb64a6a",
      "username":"admin",
      "network":"datacenter",
      "image":"conpot",
      "macAddressForContainer":"",
      "ipAddressesContainer":[],
      "container":
        {
          "id":"1a5b67e35f624d4c49b95",
          "ip":"192.168.2.20",
          "mac":"36:8B:AE:31:92:1B",
          "hostid":"36:8B:AE:31:92:1B/None"
        }
    },
    {
```

```
        "serviceid":"a4ddc2e6af4abc2ae9abb3333112d19c",
        "username":"admin","network":"datacenter",
        "image":"honeyd","
        macAddressForContainer":"",
        "ipAddressesContainer":["192.168.1.137"],
        "container":
          {
            "id":"45c7f15dc3a64747e67cb81",
            "ip":"192.168.0.82",
            "mac":"C2:79:0A:30:0E:92",
            "hostid":"C2:79:0A:30:0E:92/None"
          }
      }
  ]
}
```


POST – Create a new vHoneypot service. The honeypot (depending on the provided image) may have a forced mac address or fake a list of IP addreses. Example of a JSON payload:


```
{
   "network":"datacenter",
   "image":"conpot",
   "ips":"192.168.1.137,192.168.1.138,192.168.1.139",
   "macaddress": "00:00:00:00:00"
}
```


- `/sdn/vhoneypot/<serviceid>` – Manages a specific vHoneypot service

  GET – List the vHoneypot information (see `/sdn/vhoneypot` endpoint)

  DELETE – Remove the vHoneypot service

C.4.2 Command line

- `create-vhoneypot <username>` - Create a vhoneypot service and link it with a tenant
- `vhoneypot-delete-all` - "Factory reset"
- `vhoneypot-list` - List all vHoneypot serviceIds
- `vhoneypot-list-details` - List all vHoneypot with details
- `vhoneypot-setimage <serviceid> <repository/image>` - Associate an image with the service (pre-deployment)
- `vhoneypot-setnetwork <serviceid> <network>` - Associate a network with the service (pre-deployment)
- `vhoneypot-set-ips <serviceid> <iplist>` - Associate a list of fake ip addresses to the service
- `vhoneypot-set-mac <serviceid> <mac>` - Force the honeypot container to adopt a fixed (and provided) MAC address
- `vhoneypot-start <serviceid>` - Start a specific service
- `vhoneypot-remove <serviceid>` - Delete a specific service

C.5 Data Diode

C.5.1 HTTP endpoints

- `/sdn/datadiode` – Manages the data diode service

  GET –Lists all the data diode services. Example response:

```
{
"datadiode":
 [
  {
     "serviceid":"65d49a4f1dd067c46439ef096bb64a6a",
     "username":"admin",
     "network":"datacenter",
     "linkid": "00:0C:29:C6:0A:18/None-of:0000a0369fbc27f0",
     "direction":"from"
  }
 ]
}
```

  POST – Create a new data diode service. The JSON payload is provided below. The direction is relative to the host ("from" means blocking all packets coming from the host while "to" means blocking all the packets which the direction is the host).

```
{
  "network":"datacenter",
  "linkid":"00:0C:29:C6:0A:18/None-of:0000a0369fbc27f0",
  "direction":"from"
}
```

- `/sdn/datadiode/<serviced>` – Manages a specific data diode service

  GET – Obtain the information of a specific service (works like a filter on the data provided by the global namespace)

  DELETE – Remove the data diode service

C.5.2 Command line

- `datadiode-create <username>` - Create a data diode service and link it to a tenant account
- `datadiode-set-network <serviceid> <network>` - Set a network to a data diode service (pre-deployment)
- `datadiode-remove` - "Factory reset"
- `datadiode-list` - List all data diode service Ids
- `datadiode-list-details` - List all data diode services with extended information (network, user, link, direction, etc)
- `datadiode-remove <serviceid>` - Remove a specific data diode
- `datadiode-set-link <serviceid> <linkid> <direction>` - Set functional information on a previously created data diode service instance

- `datadiode-start <serviceid>` - Start a service (i.e. begin the network programming step)

C.6 Network Event Factory

C.6.1 HTTP endpoints

- `/sdn/nef` # – global application namespace

  GET – Shows the application configuration:

```
{
  "pt.uc.dei.atena.networkeventfactory.imp.components.
    NetworkEventFactoryManager":
    {
      "broker":"kafka.dei.uc.pt",
      "topic":"sdnevents"
    },
  "pt.uc.dei.atena.networkeventfactory.monitors.NEFDeviceMonitor":
    true,
  "pt.uc.dei.atena.networkeventfactory.monitors.NEFHostMonitor":
    true,
  "pt.uc.dei.atena.networkeventfactory.monitors.NEFTopologyMonitor":
     true,
  "pt.uc.dei.atena.networkeventfactory.monitors.NEFLinkMonitor":
    true,
  "pt.uc.dei.atena.networkeventfactory.monitors.NEFControllerMonitor
    ": true
}
```

  POST – Modifies the application configuration. This includes the broker and topic as well as enabling or disabling any monitor referenced above (JSON payload has the exact same format)

C.6.2 Web-sockets endpoints

- `/ws/nef` # – A connection to the websocket endpoint with an admin token will lead to the stream of the statistics concerning the number of events sent by each of the monitors (device events, host events, link events, topology events and controller events.

C.6.3 Command line

- `nef-event-count` - Get sent event count

Admin only

Appendix D

Research Paper

This Annex includes the produced research paper published in the scope of the first round of research publications:

Freitas, M. and Rosa, L. and Tiago Cruz and Simões, P. , "SDN-enabled virtual data diode", in Proc. of the 4th ESORICS Workshop On The Security Of Industrial Control Systems & Of Cyber-Physical Systems (CyberICPS 2018), September 2018

Presented in Barcelona, in September 2018.

SDN-enabled virtual data diode

Miguel Borges de Freitas[1], Luis Rosa[1], Tiago Cruz[1], and Paulo Simões[1]

Centre of Informatics and Systems, Department of Informatics Engineering,
University of Coimbra, Portugal
{miguelbf,lmrosa,tjcruz,psimoes}@dei.uc.pt

Abstract. The growing number of cyber-attacks targeting critical infrastructures, as well as the effort to ensure compliance with security standards (e.g. Common Criteria certifications), has pushed for Industrial Automation Control Systems to move away from the use of conventional firewalls in favor of hardware-enforced strict unidirectional gateways (data diodes). However, with the expected increase in the number of interconnected devices, the sole use of data diodes for network isolation may become financially impractical for some infrastructure operators.
This paper proposes an alternative, designed to leverage the benefits of Software Defined Networking (SDN) to virtualize the data diode. Besides presenting the proposed approach, a review of data diode products is also provided, along with an overview of multiple SDN-based strategies designed to emulate the same functionality. The proposed solution was evaluated by means of a prototype implementation built on top of a distributed SDN controller and designed for multi-tenant network environments. This prototype, which was developed with a focus in performance and availability quality attributes, is able to deploy a virtual data diode in the millisecond range while keeping the latency of the data plane to minimal values.

Keywords: Data Diode · Unidirectional gateways · Software Defined Networks · Industrial and Automation Control Systems.

1 Introduction

Industrial Automation and Control Systems (IACS) encompass a broad range of networks and systems used to monitor, manage and control cyber-physical processes in critical infrastructures, such as the power grid or water distribution facilities. The growing number of cyber-attacks against today's highly distributed IACS is raising awareness towards the need for in-depth cyber-security strategies, somehow leading to a shift back to these system's origins with the use of data diodes. When SCADA systems first appeared in the 1960's they were implemented as air-gapped islands restricted to the process control perimeter and specially isolated from corporative networks. Security was granted due to intrinsic isolation and the use of proprietary and poorly documented protocols [9].

In the 1990's, business requirements to increase productivity and performance, together and the massification of ICT technologies, broke with the previous isolated generation of IACS. Organizations began to adopt open TCP/IP

connections to link their process control and Enterprise Resource Planning (ERP) systems. Corporate management layers took advantage of this real-time data to manage plant inventories, control product quality and monitor specific process variables. It is estimated this network interconnection lead to 3-8% cost savings at large facilities [15] – however, it also brought a drastic increase on the inherent cyber-security risks, by contributing to expand the exposed IACS attack surface.

To mitigate unwanted accesses to the IACS network, middleboxes such as firewalls started to be implemented as digital barriers in the perimeter of both process and organizational networks, sometimes sitting behind a DMZ. Firewalls are often prone to configuration mistakes and are relatively accessible for exploit development by skilled individuals. In the long-term, firewalls are known to have considerable operating costs as firewall rules have to be continuously audited and maintained while firmware updates must be installed as soon as they are available [16]. The use of firewalls on IACS networks also contradicts some of their fundamental requirements: the need for real-time access to plant data, high availability and service continuity. As middleboxes, commercial off-the-shelf (COTS) firewalls introduce latency and jitter in the network, also introducing a point-of-failure (e.g., when subject to flooding attacks, throttling policies may cause service disruption).

Unlike firewalls, data diodes provide a physical mechanism for enforcing strict one-way communication between two networks. They are also known as *unidirectional gateways* since data can be securely transfered from an restricted access network (such as a process control network) to a less secure network (the corporate zone) with no chances of reverse communication. Data diodes are often built using fibre optics transceivers, through the removal of the transmitting component (TX) from one side of the communication and the respective receiver component (RX) from the opposite side [11]. This makes it physically impossible to compromise such devices to achieve reverse connectivity. Moreover, they usually do not contain firmware, requiring minimal or no configuration at all, or have minimal software supported by micro-kernels that can be formally verified [5].

Data diodes allow organizations to retrieve valuable data generated at the process level, while guaranteeing the trustworthiness and isolation of the critical infrastructure. They are the only devices receiving the Evaluation Assurance Level 7 (EAL7) grade in the Common Criteria security evaluation international standard. As a result, NIST recommends the adoption of data diodes [17].

Despite its advantages, from a security standpoint, data diode implementations come with high capital expenditure for organizations: it is estimated that for a typical large complex facility such costs can reach $250,000 while recurring support costs may ascend to values circa $50,000/year per data diode [19]. Furthermore, most data diode solutions are vendor-dependent, with the range of supported protocols strongly depending on the specific implementation – this means that many protocols on which some organizations rely upon may not be supported at all. Moreover, like any middlebox, data diodes need to be physically placed at a specific point in the network topology to be able to block network traffic, eventually requiring multiple deployments to secure dispersed network

segments. Considering such shortcomings, many organizations may not be willing to invest in devices that are not future-proof or lack flexibility, fearing they may become outdated by the time their break-even point is reached.

To deal with the inherent limitations of existing solutions, we propose using Software Defined Networking (SDN) and Network Function Virtualization (NFV) to implement a cost-effective data diode. SDN aims at shifting the network equipment control plane functionality to a logically centralized entity – the network controller. In SDN, network switches are turned into "dumb" devices whose forwarding tables are updated by the network controller, using open protocols such as Openflow [13]. NFV provides a way to decouple network equipment functionality in several chained Virtual Network Functions (VNFs), which may be hosted in dispersed infrastructure points-of-presence.

SDN can be leveraged to implement innovative network security approaches: the network controller has a global view of the network topology graph, has real-time state awareness over all allowed network flows and can modify the network state by means of a proactive (preinitializing flow rules) or reactive (deciding upon packet arrival) approach. For such reasons, an SDN-based data diode could provide an alternative to both firewalls and conventional appliances. Note that to efficiently forward network packets, general purpose network switches contain forwarding tables called TCAMs (*ternary content-addressable memory*) which are able to perform an entire table lookup in just one clock cycle [18]. Hence, an SDN-enabled virtual data diode could effectively block traffic at Layer 2, avoiding the typical latency imposed by firewall middleboxes. Vendor lock-in, management complexity and deployment issues are also mitigated due to the use of open protocols, the existence of a single managing interface to control the overall network and the removal of placement restrictions imposed by hardware appliances. SDN also helps future proofing virtual data diode implementations: the data diode application can be easily adapted to support new protocols, and/or new network functions can be added to the network via NFV.

The remainder of this paper is organized as follows. Section 2 provides a review of the major COTS data diode products, together with an overview of the main challenges for protocol support in unidirectional communications. Section 3 explains how SDN can be leveraged to implement a functional data diode. Section 4 presents our proof-of-concept (PoC) prototype: a simple SDN data diode that is able to support the UDP protocol, implemented in a distributed network controller environment and geared towards performance and availability. Finally, Section 6 provides a wrap-up discussion and concludes the paper.

2 Data diodes for IACS security

Data diodes are devices that restrict the communication in a network connection so that data can only travel in a single direction, having borrowed their name from electronic diode semiconductors. Although different hardware implementations exist, supporting different physical layers (e.g. RS-232, USB, Ethernet), most make use of optical couplers to guarantee physical isolation. The trans-

mitter side of a data diode converts electrical signals to optical form using light emitting diodes (LEDs), while at the receiving end photo-transistors convert the optical data back to electrical form [8]. It is the physical air-gap in the optical-coupler that makes data diode devices so secure and appealing in the critical infrastructure context.

In IACS, data diodes are often deployed to isolate specific network domains or between corporate and process control networks, to support the unidirectional transfer of historian data, HMI screen replication, or for one-way telemetry (operational data, security events, alarms and syslog). Data diodes are commercially available in two different form factors: single-box solutions and PCI express cards [10]. The former category may also encompass single-box or split-device variations, in which a component is deployed at each side of the connection.

Despite the similarities in the key isolation mechanism, commercially available solutions differ significantly in terms of supported services and protocols. Data diodes have to make use of additional software components for each side of the unidirectional link to be able to support TCP/IP-based SCADA protocols, such as MODBUS/TCP, Ethernet/IP and DNP3. Such protocols were designed for bidirectional operation, relying on a three-way handshake and requiring continuous acknowledgments between communication peers. In [6] the TCP workflow in unidirectional links is explained along with the presentation of a design for a unidirectional gateway for IACS applications (Figure 1).

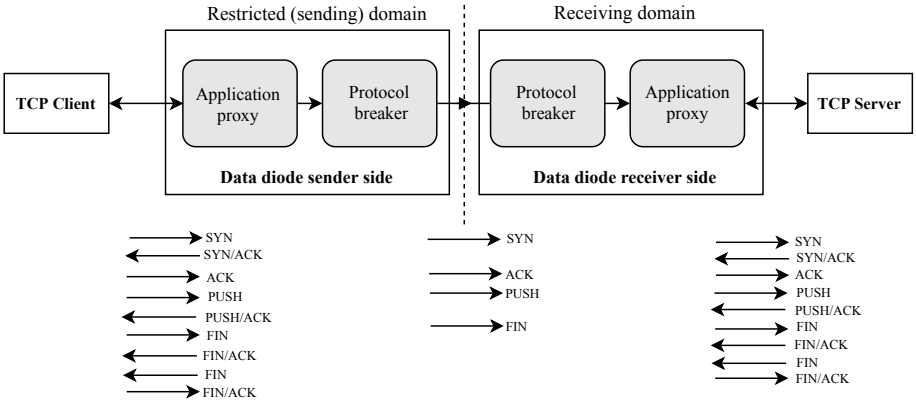


Fig. 1: TCP workflow in uni-directional gateways (adapted from [6]).

The architecture includes two different components at the edge of the unidirectional link: (i) an application proxy and (ii) a protocol breaker. The former is responsible for acting as a proxy for TCP connections. In the sender side of the data diode, the application proxy operates as a TCP server, automatically responding with SYN/ACK, PUSH/ACK and FIN/ACK to any SYN, PUSH or FIN packets sent by the TCP client. Any packet generated by the TCP client is

forwarded by the application proxy to the unidirectional link. On the receiving end, the application proxy simply emulates the TCP client forwarding any received packets. The protocol breaker component acts as a middleware for packet encapsulation for protocols that do not require acknowledgments (e.g. UDP). It can also be used to provide confidentiality within the unidirectional link or to apply forward error correction to the data transfer.

2.1 Data Diode Products

There are diverse commercial data diode solutions in the market, depending on the specific use case and protocol support. Table 1 provides a summary of the three most notorious products in the context of IACS. Next, we provide a brief review of those products, based on publicly available documentation.

Table 1: IACS commercial data diodes.

Company	**Owl CyberDefense**	**Fox-IT**	**Waterfall**
Form Factors	1U rack mount, DIN rail, PCIe cards	1U rack mount	Modular designs: gateway pairs (1U), single box(1U), DIN rail
Bandwidth	10Gbps	1.25Gbps	1Gbps
IACS Applications	Rockwell, OSIsoft PI, Schneider Electric	OSIsoft PI	OSIsoft, GE, Schneider Electric, Siemens, Emerson, Areva, Honeywell, AspenTech, Scientech, Rockwell
IACS Protocols	Modbus, OPC	Modbus, DNP3, OPC, ICCP	OPC DA/HDA (backfill)/UA, A&E, DNP3, ICCP, Siemens S7, Modbus, Modbus Plus, IEC 60870-5-104, IEC 61850
CC Certification	EAL4	EAL7+	EAL4+

Owl CyberDefense provides the DualDiode^{TM} technology as part of the company cross domain solution portfolio. Owl's data diodes make use of a hardened Linux kernel, providing optical separation and a protocol breaker that converts all packets to non-routable Asynchronous Transfer Mode (ATM) cells, also supporting data transfers up to 10Gbps [14]. Protocol support includes TCP/IP connections, UDP, Modbus and the OPC family, as well as historian solutions from Rockwell Automation, Schneider Electric and OSIsoft. Latest revisions of DualDiode^{TM} Network Interface cards received CC EAL4 certification.

Fox-IT's DataDiode^{TM}, is compliant with the highest level of CC certification: EAL7+ [3]. It implements full protocol break capabilities and uses a single

optical fiber strand, together with custom optoelectronics designed for one-way operation. Being a firmware-less device, it has no configuration or local state, relying on proxy servers deployed on each side of the connection. These proxies implement several techniques for error detection and increased reliability, using metadata for lost packet detection (supported by proxy-level logs, for manual retransmission), forward error correction codes and heartbeat mechanisms [4]. In government editions, the device includes an anti-tampering mechanism [3]. The Fox-IT data diode is able to achieve 1.25Gbps in the link layer, although the actual speed is lower due to the proxy servers. It claims to support Modbus, DNP3, OPC and ICCP protocols along with file transfers, SMTP, CIFS, UDP and NTP [7]. The OSIsoft PI Historian is also supported.

Waterfall Security Solutions provides data diode appliances in multiple form-factors, including split-pair, single-box and DIN rail versions, based on a modular combination of hardware and software[22]. Such unidirectional gateways include a TX-only module (containing a fiber-optic laser), a fiber optic cable, an RX module (optical receiver), together with host modules that gather data from industrial servers and emulate different protocols and industrial devices. The latter are provided either as standalone physical modules or virtual machines. Popular industrial applications/historians are supported (e.g. Osisoft PI System, GE iHistorian, Schneider-Electric Instep eDNA), as well as a long list of industrial protocols (e.g. Modbus, DNP3, OPC, Modbus Plus [20]). Devices support up to 1Gbps data transfers and are certified EAL4+. The company recently announced a reversible hardware-enforced unidirectional gateway whose direction can be controlled by software, using a schedule or exception-based trigger mechanism [21].

3 Leveraging SDN to virtualize the data diode

The OpenFlow protocol is a Layer 3 network protocol that gives access to the forwarding plane of a network switch over the network. It enables network controllers to determine the path of network packets across the switch fabric. The protocol works on top of TCP/IP although the communication between the controller and the switch can also be configured to make use of the Transport Layer Security (TLS) protocol. The protocol works in a *match-action* manner: when a packet arrives at a switch port, the switch starts by performing a table lookup in the first flow table to match the packet headers against the set of flow rules installed in the switch. If a match is found, the switch applies the instruction set configured in the flow rule. In case of a table miss, the corresponding packet action depends on the table configuration: the packet can be forwarded to the controller for further processing (using *Packet-In* messages), can be moved further on the flow table pipeline, can have header fields re-written or can simply be dropped [13]. The match fields in an OpenFlow flow table comprise fields ranging from Layer1 to Layer4 (Table 2) permitting a fine-grained control over the packet identification and ultimate destination.

Table 2: The OpenFlow flow table match fields [13].

Match Field	Description
IN_PORT	Ingress Port (physical or a switch defined logical port)
ETH_DST	Ethernet destination MAC address
ETH_SRC	Ethernet source MAC address
ETH_TYPE	Ethertype of the packet payload
IPv4_SRC	Source IP address
IPv4_DST	Destination IP address
IPv6_SRC	Source IP address (IPv6 format)
IPv6_DST	Destination IP address (IPv6 format)
TCP_SRC	TCP source port
TCP_DST	TCP destination port
UDP_SRC	UDP source port
UDP_DST	UDP destination port

Taking into account the workflow of a packet reaching an OpenFlow enabled switch, we identify three different approaches for an SDN-based virtual data diode: proactive; reactive; and NFV-assisted.

3.1 Proactive data diode

A proactive data diode is an SDN unidirectional gateway implementation that takes advantage of OpenFlow's proactive flow rule instantiation It is the simplest and most limited implementation since it can only support applications that rely on the UDP protocol. Considering two networks with different degrees of classification (cf. Figure 3), the network controller installs (in advance) two rules in the restricted (sending) domain uplink switch (cf. Table 3). One of the rules instructs the switch to drop any packets entering the switch and originating at the switch port that is connected to the receiving network uplink switch. The other rule forwards any packets entering the remaining switch ports to the receiving domain uplink switch port.

Further limitations can be applied in the second flow rule to limit the devices from the receiving domain network that are allowed to unidirectionally transfer data, using the IN_PORT, ETH_SRC and IPv4_SRC/IPv6_SRC match fields. For the proactive data diode to support TCP applications, the sending machine has to encapsulate the packet into an UDP packet. Alternatively, flow rules can be installed on the switch to set the UDP source and destination ports (and replace the TCP source and destination fields) to any TCP packets entering the switch. In both cases, in order to support the TCP protocol, additional software is required in the receiving machine to disassemble the received packets into usable data. For this type of virtual data diode, only the uplink switch in the restricted network domain needs to support OpenFlow. The remaining sections of both networks may still rely on traditional network architectures.

3.2 Reactive data diode

Instead of installing rules in the up-link network switch, the reactive data diode instructs the switch to forward any received packet headers to the network controller for further processing (Table 4). The network controller can check if the received packet comes from the receiving domain (by looking up the input port) and simply instruct the switch to drop the packet. Similarly, it can instruct the switch to forward the packet if it originates from the restricted (sending) network domain.

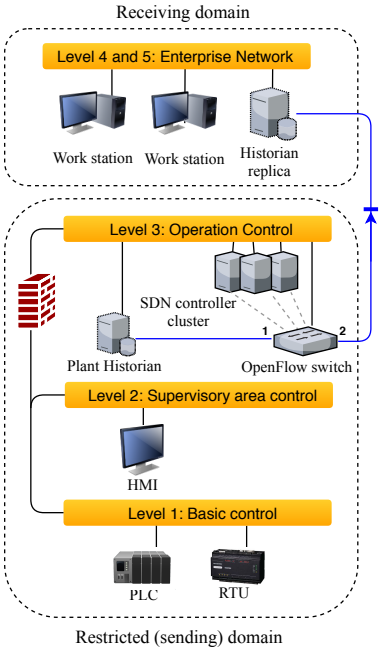


Fig. 2: SDN-enabled virtual data diode.

Table 3: Proactive data diode flow table.

Table	Match Fields	Action
0	in_port=1	output:2
0	in_port=2	drop

Table 4: Reactive data diode flow table.

Table	Match Fields	Action
0	in_port=1	output:controller
0	in_port=2	output:controller

Using a reactive approach, the network controller has greater flexibility since it can add support to the TCP protocol. It can behave as *an application proxy* for TCP connections implementing a workflow similar to the one in Figure 1. TCP acknowledge packets can be faked by the controller and outputted via a switch port to the host establishing the connection. Hence, the TCP protocol can be supported while still only allowing unidirectional communications as long as an application proxy is able to perform the same workflow in the low-priority network. Furthermore, there are some cases in which bi-directional communication between both networks is required or should be temporarily enabled (e.g. an application that relies on TCP for initial connection establishment). The network controller can be programmed in such a way that bi-directional communication is enabled in certain situations. Thus, it is possible to emulate the behavior of the Waterfall's reversible data diode. Despite the provided flexibility, this approach

introduces latency in network flows (due to additional packet processing) and the network controller is vulnerable to flooding attacks. Packets originating in the receiving network domain will not be forwarded to the hosts on the restricted domain without the permission of the controller. Nevertheless, hosts on the receiving domain are able to flood the OpenFlow switch with packets destined to the restricted network. Those packets are ultimately redirected to the controller, causing a denial of service which disrupts the unidirectional communication that is expected to happen in the reverse direction.

3.3 NFV-assisted data diode

This approach requires SDN support at the edge of the restricted (sending) network, as well as a virtualization infrastructure containing a virtual OpenFlow switch (e.g. OpenvSwitch). It represents a combined approach where the processing step is supported by virtualized hosts close to the uplink OpenFlow switch and directly accessible to the SDN network. Network traffic originating in the restricted domain with the receiving network as destination is offloaded by the first OpenFlow switch to a dedicated virtual host. This virtual host can either be a virtual machine or an application container with two virtual Ethernet interfaces: one for receiving network packets and another for the output of packets. TCP emulation is performed within the virtual host by automatically generating acknowledgment packets for the three-way handshake and subsequent TCP transfers. Packets that are meant to be sent to the low priority network are chained from the input virtual interface to the output virtual interface (e.g. using IPtables).

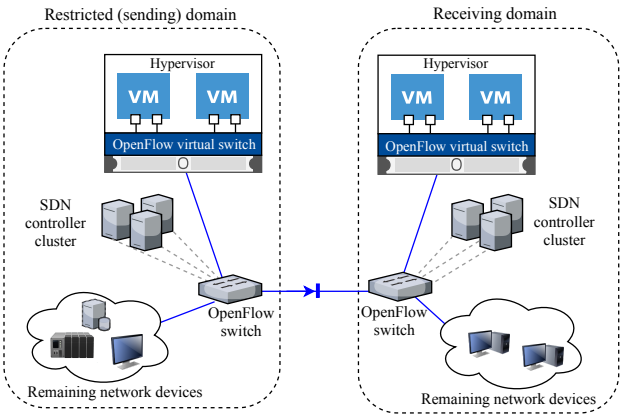


Fig. 3: NFV assisted SDN virtual data diode.

Flow rules are proactively installed by the network controller in the uplink switch to: (i) drop any packets coming from the receiving network; (ii) forward any packets from the virtualization host (output port) to the switch port connected to the receiving network; (iii) forward any other packets to the virtualiza-

tion host input port. In the receiving network domain, a TCP emulation proxy host should also exist and a similar approach can be applied. This data diode implementation avoids flooding attacks against the control plane while keeping the flexibility of the reactive design approach. Protocol support can easily be added to the virtual application proxy. The global network topology available at the controller can be used to automatically find the path (sequence of ports) leading to the virtual host. Moreover, if a layer of orchestration is added to the controller, it can continuously monitor the state of the virtual host and request the creation of a new one in case of failure (adjusting the flow rules to respect the new virtual ports).

4 Proof-of-Concept Virtual Data Diode Prototype

In the context of IACS, availability, performance and the need for real-time operation are the key design system attributes. As such, we developed our PoC virtual data diode using distributed SDN controllers, so that the control plane itself does not represent a single point of failure in the overall system operation. Distributed controllers are multi-node architectures where each OpenFlow switch maintains an active connection to one of the controller nodes (the master node) but is configured to use redundant connections to other nodes (slaves), in the case of master node failures. Although many network controller projects exist, only a small minority is distributed [12]. Among those, we selected the Open Network Operating System (ONOS) because it matches well into the critical infrastructure use-cases: high throughput (up to 1 M requests/second), low latency (10-100 ms event processing) and high availability (99.99% service availability)[2]. Our PoC virtual data diode uses a proactive approach regarding flow rule instantiation. Flow rules are installed from a dashboard containing the global topology graph of the network. Using this approach, the OpenFlow switches are still able to virtualize a data diode even in the case of an hypothetical full control plane failure. To increase performance, the data diode does not rely on any controller external interfaces. It was implemented directly in the application (using its OSGi services), extending its external interfaces (REST, command-line and websockets). Figure 4 presents the PoC architecture.

By default there is no connectivity between hosts in the SDN network. The *Proxy ARP* application (ONOS-bundled) proactively installs rules in the switch fabric to forward any ARP packets to the controller so the topology graph and host location can be computed. The developed *Network Manager* application relies on intent-based networking to provide connectivity between a set of hosts in the network. Intents are ONOS high-level abstractions (protocol independent) that allow applications to define generic connectivity policies that are translated internally to flow rules. For each host pair, the host-to-host intent results into two installed rules (with fixed priority) using the *in_port*, *eth_src* and *eth_dst* as match-fields and outputting to a port leading to a path to the host location. ONOS monitors the network state and any installed intents: if a network switch is unavailable and a redundant path between hosts exists, a new set of flow rules

is generated and installed, keeping the intent active. By ensuring selected host connectivity, the *Network Manager* application creates logical subsections in the overall topology graph, providing the basis for multi-tenancy.

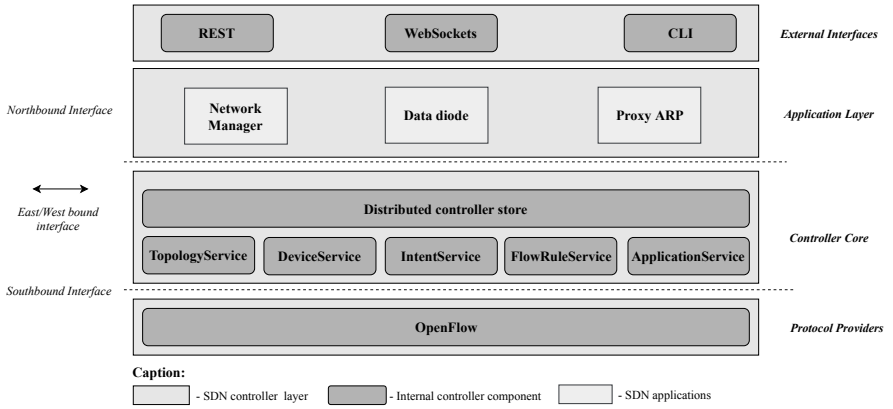


Fig. 4: Architecture of the Virtual Data Diode PoC.

The *Data Diode* application then uses the information stored by the *Network Manager*. When a deployment is requested, given a topology edge link and the network name, the application finds the connection point (host-switch/port) in the graph and requests the *Network Manager* the list of hosts belonging to that network. The application then installs one rule per network host-pair in the edge switch (identical to one of the rules installed by the *Network Manager*) with the action field set to *Drop*. Those flow rules have a higher priority field than the rules defined by the *Network Manager* application superseding them. A workflow similar to the one depicted in Table 3 was not followed in the implemented prototype, in order to avoid binding physical ports to data diode deployments and preserve multitenancy support. Additionally, the *Data Diode* implements a monitor that asynchronously receives any events produced by the network (*Network Manager* application) – its purpose is to install new rules to enforce the diode behavior for each new host.

5 Evaluation

This section discusses the experimental evaluation of our PoC virtual data diode.

5.1 Experimental Testbed

Figure 5 ilustrates the testbed and network topology used for the validation of the virtual data diode prototype. It consists of a single OpenFlow switch controller by a three node ONOS cluster. The OpenFlow switch was running OpenvSwitch (CentOS 7) in a COTS server (Dell Poweredge R210), with six available gigabit Ethernet interfaces. The server was configured with Intel DPDK

for increased network performance (bypassing the Linux Kernel and promoting direct memory access using hugepages and the VFIO universal IO driver). The switch configured to use the three controller plane nodes, connects to the master node via an off-band management network not accessible to the hosts in the SDN network. The three controller nodes were CentOS 7 virtual machines, each with 4GB of RAM. The network hosts are composed by an Environmental Monitoring Unit (EMU) and two Modbus TCP agents. The EMU is an arduino-based board with built-in Ethernet ASIC (Freetonics EthertTen), containing a DTH11 sensor and an electromechanical relay. The temperature, humidity and relay state values are kept updated in three holding registers, and made available in the SDN network via the Modbus TCP protocol.

The Modbus TX and RX hosts are virtual machines with gigabit ethernet configured in passthrough mode. Their role is to emulate the behavior application proxies and protocol breakers found in commercial data diodes (cf. Figure 1). The TX agent queries the EMU holding registries, serializes the data into the pickle format and sends it through the UDP protocol to the RX agent. This RX agent behaves as the EMU device on the other side of the network. It desserializes the received data, updates the internal registries and exposes a Modbus TCP server. A virtual data diode was deployed (from the SDN controller) in the edge link connecting the switch to the RX agent. Thus, the connection between both agents was considered unidirectional (TX $\rightarrow$ RX only).

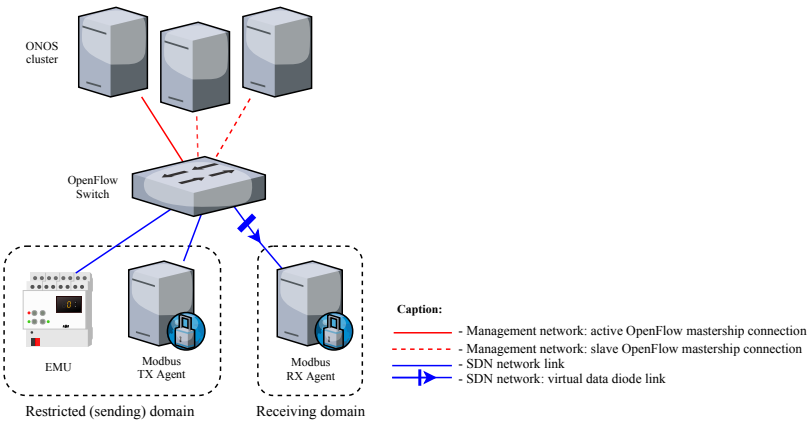


Fig. 5: Experimental testbed.

5.2 Validation and Lessons Learned

The functional validation of the virtual data diode was achieved recurring to the Netcat tool: the RX agent was configured as an UDP server while the TX agent acted as a client and vice-versa. We confirmed that in the former case packets were able to flow while in the last no communication occurred. The non-functional validation focused on assessing the prototype performance. Experiments focused on three aspects:

(a) the effect of the data layer on the latency of Modbus TCP readings;
(b) the overall network performance of the data plane;
(c) and the deployment latency of the virtual data diode.

For (a) we designed a test consisting of an increasing number of sequential reads of ten EMU holding registries. For the TX agent we removed the ability to process and packetize the obtained data and measured the time immediately before and after each query. The measured times should be taken as the base values for reading latency. For the RX readings, the time was recorded right after data has been desserialized and updated in the agent context. Furthermore, a counter was increased upon receiving a reading from the TX agent. Total test duration was computed using the temporal instant before the first query by the TX agent as starting time. Both machines were synchronized via NTP before performing the test and each test was repeated five times. Table 5 summarizes the obtained latencies (and percentage of failed readings). Confidence intervals were calculated using a t-student distribution with a 95% confidence interval.

Table 5: Latency effect of the data layer on Modbus TCP readings.

Modbus Agent	**Number of Queries**	**Time (s)**	**Failed Reads (%)**
TX	1	0.067 ± 0.139	-
	10	9.889 ± 0.640	-
	100	111.045 ± 0.331	-
	500	566.654 ± 0.558	-
RX	1	0.654 ± 0.344	0
	10	10.185 ± 0.777	0
	100	111.820 ± 0.897	0
	500	567.679 ± 0.549	0.360 ± 0.444

It is possible to conclude that even though the added latencies show a cumulative effect with respect to the number of readings (almost defining a linear trend) the latency increase is almost negligible. For 500 EMU readings, the additional processing by the agents and the subsequent network transfer only delays the overall reading time by 1 second. It is also possible to see that, as the number of queries increases, we start noticing a minimal amount of readings not reaching the RX agent – although being reported as sent by the TX agent. This can be explained by the no-guarantee nature of the UDP protocol. While this problem could be mitigated by adding error correction mechanisms to the unidirectional data packets or sending the same packet multiple times, in experiment (b) we analysed the effect of the sender/receiver buffer size on packet loss. This experiment also measured the maximum bandwidth of the data diode link.

For asssessing (b), iPerf was used to limit the TX sender bandwidth at values ranging from 10 Mbps to the maximum theoretical value of the link (1 Gbps) while changing the sender buffer size (100-6000 KB). The virtual data diode was disabled during this test, since Iperf requires an initial TCP connection. Measurements show that the buffer size plays a significant role on the packet

loss, since it affects the total number of packets that can be sent in a single transfer (cf. Figure 6).

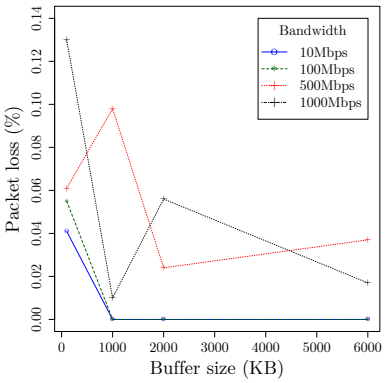


Fig. 6: Percentage of lost packets vs. bandwidth and write buffer size.

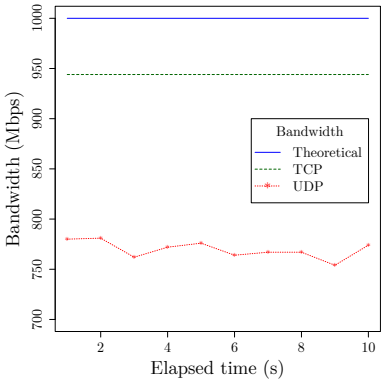


Fig. 7: UDP bandwidth vs. TCP and Theoretical bandwidth.

If the bandwidth is known beforehand, both agents can be optimized for minimal packet loss. This is important in IACS scenarios, since SCADA traffic patterns tend to be predictable, with stable network topologies [1]. Regarding the stress test on the data diode link, we started by performing a TCP test. The bandwidth achieved by TCP is expected to be higher than the actual bandwidth of the UDP transfer since it optimizes the transfer window size during the transfer. We took the measured value (944 Mbps) as the reference for the actual bandwidth. The maximum bandwidth using UDP was 769.7 ± 7.4 Mbps (cf. Figure 7), a value in line with some commercial switches, despite our software-based testbed.

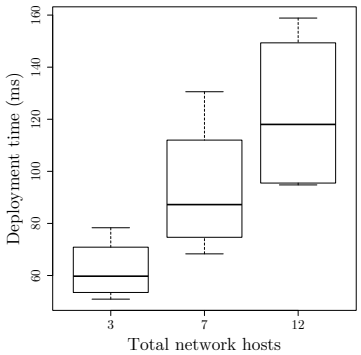


Fig. 8: Virtual data diode deployment times *vs* number of network hosts.

Table 6: Virtual data diode deployment times depending *vs* number of network hosts

Network hosts	**Deployment time (ms)**
3	62.188 ± 14.810
7	93.330 ± 33.345
12	122.418 ± 39.790

For the (c) experiment, we measured how the deployment times of the virtual data diode varied, accordingly to the number of network hosts. Hosts were

"faked" by changing the MAC address and the IP address of one of the machines, followed by the generation of ARP packets. Upon detection in the network controller, those fake hosts were added to the previously created network. A controller command-line command was introduced in the *Data diode* application to deploy and remove the virtual-data diode in a loop, while collecting the elapsed time. Table 6 and Figure 8 present the results. Although the deployment times increase with the number of network hosts, it is in the millisecond range. A small value considering that for n hosts, n-*1* flow rules have to be installed and the datastore has to be consistently synchronized between all the controller nodes.

6 Conclusion

Current trends, such as Industry 4.0 and Internet of Things are evolving industrial control networks towards ubiquity, moving away from the traditional monolithic and self-contained infrastructure paradigm, in favor of highly distributed and interconnected architectures. In this perspective, the use of data diodes provides a convenient way to isolate mission-critical network domains, while still allowing for relevant information (i.e., telemetry) to be accessed from the outside. However, as the number of interconnected devices increases, the costs of multiple physical data diodes may become impractical for organizations.

To deal with the inherent limitations of traditional implementations, we proposed the virtual data diode concept, which leverages the benefits of SDN and NFV. This concept was demonstrated and evaluated by means of a proof-of-concept prototype, designed with performance and availability in mind. The use of proactive flow rule instantiation removes the complete dependency on the control plane, allowing the virtual data diode to use the available switch bandwidth. The use of a distributed controller provides reliability and continuous operation in case of controller node failures. Prototype evaluation measurements recorded virtual data diode deployment latencies in the millisecond range, with minimal latency in the link layer. Even stressing the switch to its full rate capacity (with much higher values than the ones typically found in IACS), packet loss in the link was minimal. While not providing the same security levels of physical data diodes (it is a software implementation), the virtualized version still compares favorably with diode alternatives, such as firewalls, while maintaining functional equivalence to its physical counterpart.

Acknowledgements

This work was partially funded by the ATENA H2020 Project (H2020-DS-2015-1 Project 700581) and Mobiwise P2020 SAICTPAC/0011/2015 Project.

References

1. Barbosa, R.: Anomaly detection in SCADA systems : a network based approach. Ph.D. thesis, University of Twente (2014), doi:10.3990/1.9789036536455

2. Berde, P., Gerola, M., et al.: ONOS: towards an open, distributed SDN OS. Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14 pp. 1–6 (2014), doi:10.1145/2620728.2620744
3. FoxIT: Fox DataDiode Data Sheet (2018), https://www.fox-it.com/datadiode/downloads/
4. FoxIT: Fox IT FAQ. Online (2018), https://www.fox-it.com/datadiode/faq/
5. Genua: Data Diode cyber-diode. Brochure (2018), https://www.genua.de/fileadmin/download/produkte/cyber-diode-flyer-en.pdf
6. Heo, Y., et al.: A design of unidirectional security gateway for enforcement reliability and security of transmission data in industrial control systems. In: Int. Conf. on Advanced Communication Technology (2016), doi: 10.1109/ICACT.2016.7423372
7. Jeon, B.S., Na, J.C.: A study of cyber security policy in industrial control system using data diodes. In: 18th International Conference on Advanced Communication Technology (ICACT). p. 1 (jan 2016), doi:10.1109/ICACT.2016.7423373
8. Jones, D.W.: RS-232 Data Diode - Tutorial and reference manual. Tech. rep., United States (2006)
9. Mckay, M.: Best practices in automation security (2012), doi:10.1109/CITCON.2012.6215678
10. Mraz, R.: Data Diode Cybersecurity Implementation Protects SCADA Network and Facilitates Transfer of Operations Information to Business Users. Presentation (2016)
11. Okhravi, H., Sheldon, F.T.: Data Diodes in Support of Trustworthy Cyber Infrastructure. In: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research. pp. 23:1—-23:4. CSIIRW '10, ACM, New York, NY, USA (2010), doi:10.1145/1852666.1852692
12. Oktian, Y.E., et al.: Distributed SDN controller system: a survey on design choice. Comp. Networks **121**, 100–111 (2017), doi:10.1016/j.comnet.2017.04.038
13. Open NF: OpenFlow Switch Specification Version 1.5.1 (Protocol v. 0x06) (2015), https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf
14. Owl Cyberdefense: Learn about data diodes. Online (2018)
15. Peterson, D.G.: Air Gaps Dead, Network Isolation Making a Comeback. Online, http://www.digitalbond.com/blog/2011/07/19/air-gaps-dead-network-isolation-making-a-comeback/
16. Scott, A.: Tactical Data Diodes in Industrial Automation and Control Systems. Tech. rep., United States (2015)
17. Stouffer, K.A., et al.: NIST SP 800-82 rev2. Guide to Industrial Control Systems (ICS) Security: SCADA Systems, DCS, and Other Control System Configurations Such As Programmable Logic Controllers (PLC). Tech. rep., USA (2015)
18. Sun, Y., Liu, H., Kim, M.S.: Using TCAM efficiently for IP route lookup. In: 2011 IEEE Consumer Communications and Networking Conference, CCNC'2011. pp. 816–817 (2011), doi:10.1109/CCNC.2011.5766609
19. Waterfall Security: Unidirectional Security Gateways vs. Firewalls: Comparing Costs. Tech. rep., Israel (2012)
20. Waterfall Security: Unidirectional Security Gateways (2018), https://static.waterfall-security.com/Unidirectional-Security-Gateway-Brochure.pdf
21. Waterfall Security: Waterfall FLIP (2018), https://waterfall-security.com/wp-content/uploads/Waterfall-FLIP-Brochure.pdf
22. Waterfall Security: Waterfall WF-500 product datasheet. Product Datasheet (2018), https://waterfall-security.com/wp-content/uploads/WF-500-Data-Sheet.pdf